

Agent*j*

Java Agent Framework for NS2

Ian J. Taylor

September 15, 2004

Contents

I	Introduction, Overview and Installation	1
1	Introduction	5
1.1	Motivation for Agentj	5
1.1.1	Investigation into P2P	6
1.1.2	The sensors	7
1.2	Overview of the Agentj Architecture	7
1.2.1	Agentj and the JNI PAI Interface	9
1.3	The GAP Interface and P2PS	10
1.3.1	The GAT	11
1.3.2	The GAP Interface	11
1.3.3	P2PS	12
1.4	Filling in the Gaps	13
1.5	Conclusion	14
2	Installing the Agentj Toolkit	15
2.1	Downloading the Pieces	15
2.2	Installing the Protolib NS2 Binding	16
2.3	Installing Agentj	17
2.4	Environment Variables	18
2.5	Installation into NS2	19
2.5.1	Building NS2	22
2.5.2	What's included in the Agentj Release?	34
2.6	Configuration	35
2.7	Agentj Logging	35
2.7.1	AutoLog Overview	35
2.7.2	AutoLog Discovery	36
2.7.3	Example XML Configuration	37
2.8	Conclusion	37

II	Agentj Design and Implementation	39
3	Protolib	43
3.1	An overview of Protolib	43
3.2	Protolib Structure	44
3.3	Conclusion	47
4	The PAI Interface	49
4.1	Overview of PAI	49
4.2	Programming PAI	49
4.3	Using PAI within NS2: The C++ Side	49
4.4	Ns2 Agents	51
4.5	PAI Agents and Protolib	54
4.6	Conclusion	54
5	Agentj: Java Agents in NS2	55
5.1	Agentj Software Overview	55
5.1.1	Creating Agentj Nodes	56
5.1.2	Inter-Agentj Communication	57
5.2	Agentj Implementation	59
5.2.1	Organization of Agentj Classes	60
5.2.2	Key Agentj Classes	61
5.2.3	The Java PAI interface	64
5.3	Conclusion	66
III	Using Agentj	69
6	Using Agentj	73
6.1	Invoking Java Agents from NS2 Agents	73
6.2	Creating and Attaching a Java Agent	76
6.2.1	The TCL Side	76
6.2.2	The Java Side	78
6.3	Changing the Command Delimiter	79
6.3.1	The TCL Side	80
6.3.2	The Java Side	81
6.4	Conclusion	82
7	Advanced Agentj	85
7.1	Agentj and PAI	85
7.1.1	Using the Java PAI Interface in Ns2 Java Objects	85
7.2	Example 1: Sending Data From One Node to Another	86

7.2.1	The TCL Side	86
7.2.2	The Java Side	87
7.3	Example 2: Using the Trigger Mechansim	91
7.3.1	The TCL Side	91
7.4	Example 3: Sending Data Using Multicast	93
7.4.1	The TCL Side	93
7.4.2	The Java Side	95
7.5	Conclusion	98

Part I

Introduction, Overview and Installation

In this part, we given an introduction and overview of **Agentj** including the underlying technoluigies that **Agentj** encapsulates. We then provide a detail account of how to install **Agentj** into the NS2 simulation environment for the simulation of Java distributed applications.

Chapter 1

Introduction

This chapter gives a background into the motivation behind the development of the **Agentj** framework. There is an accompanying manual [1], which describes a use of the **Agentj** framework for simulating P2P networks within the NS2 [12] simulation environment. **Agentj** provides users with the ability of simulating both real-world Java applications within NS2. It supports a subset of the common transport protocols used by applications e.g. UDP, TCP, Multicast etc and enables these protocols to be used within NS2 as if the application was running within the real Internet. The focus for **Agentj** is to simplify the process of simulating applications within NS2 and therefore native familiar Java interfaces are provided so that the application only has to undergo minimal changes in order to be simulated within NS2.

Agentj also supports C++ applications but the applications would have to program to the interfaces we provide via the PAI or Protolib toolkits. Unlike the **Agentj** Java implementation the C++ are not the same as the native socket implementations and therefore recoding would be required. In Java, an application simply needs to change package name to use **Agentj**, and the interfaces to UDP/TCP thereafter are identical. Therefore, often a find/replace on "java.net" to "pai.net" suffices and a Java application can be simulated using **Agentj** within NS2. We prove by providing the P2PS implementation, which works exactly in this way. P2PS remains the same but we've plugged in the **Agentj** communications layer.

1.1 Motivation for **Agentj**

The main developer of **Agentj**¹ is working with the Scalable, Robust Self-Organizing Sensor (SRSS) systems group in NRL, which is investigating and

¹Ian Taylor, email: ian.j.taylor@cs.cardiff.ac.uk

modelling, using network simulation tools, lightweight network application discovery mechanisms suitable for application in mobile sensor systems. The mobile sensors are envisioned to leverage self-organizing computer communication networks based on Mobile Ad-hoc Networking (MANET) routing protocols which operate using wireless communication links and have no centralized administration or control.

Each node in a MANET network participates in the discovery of a route and therefore low-level routing protocols are paramount to the overall behaviour. However, it is anticipated that middleware network services beyond routing will be required to facilitate autonomous self-organization of sensors and their various related data collection, processing, and reporting functions.

The complexity of middleware approaches being considered and examined range from utilization of simple, organic network services which might be provided by the network layer (network name/address resolution, IP multicast, ANYCAST) to potentially heavy-weight, highly stateful, complex agent-based architectures. The focus of this task will be lightweight (minimally complex) middleware discovery mechanisms and services which can facilitate publish and subscribe relationships among a set of sensor application peers participating in an SRSS network. The context of highly dynamic, possibly mobile, networking will place special challenges on such protocols ability to perform peer neighbor and service discovery and to maintain that information in the face of node outages and/or relocation within the network.

1.1.1 Investigation into P2P

To satisfy these goals, the SRSS project has been looking into the use of lightweight peer-to-peer (P2P) solutions for dynamically discovering and connecting the mobile sensor nodes. P2P middleware attempts to create a virtual overlay [2] over the existing Internet to enable collaboration and sharing of resources. Further, recent P2P approaches have been designed to connect individual users using highly transient devices and computers living at the edges of the Internet (i.e., behind NAT, firewalls etc) [3]. Therefore, **Agentj** provides the framework and other systems e.g. P2PS, provide the P2PS techniques that run within NS2 using **Agentj**. For a discussion of why P2PS was chosen instead of other middleware approaches (e.g. Jxta), see [1].

The P2P approach is interesting to the SRSS group because mobile sensors within wireless networks exhibit similar types of behaviour as Internet peers, but are hosted within an even more hostile environment; that is, nodes are disappearing/reappearing frequently, data rates are continuously changing as the sensors move away from the wireless hubs and other factors, such as battery strength, which can affect the type of role the sensor can play within

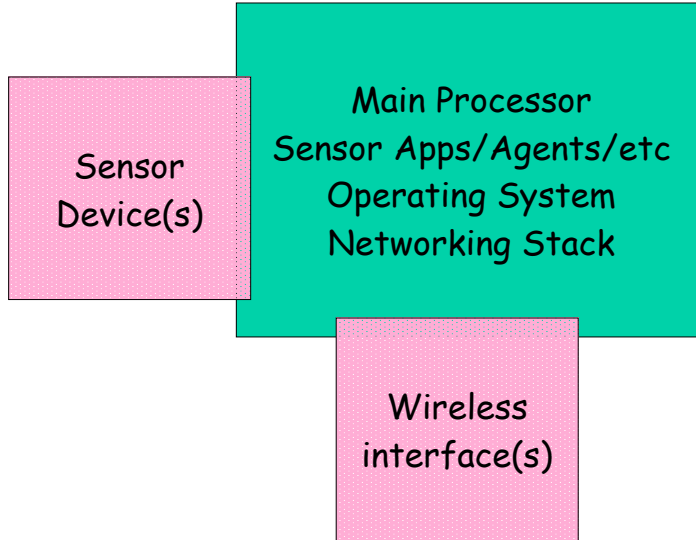


Figure 1.1: The components of a wireless sensor within an SRSS network.

the network. It could be argued that a mobile sensor application working in such an environment provides an excellent stress test for the P2PS protocols employed as it is far more dynamic than a conventional Internet application.

1.1.2 The sensors

The actual sensors are relatively simple devices that consist of a CPU, a data collection mechanism e.g. ADC convertor for audio, images etc and a wireless network card for communication across the MANET network to other participating nodes in the community.

1.2 Overview of the **Agentj**Architecture

Agentj implements a Java framework for plugging in Java applications (e.g. SRSS mobile node behaviour) and middleware, such as the GAP and its bindings e.g. P2PS. The GAP interface and P2PS middleware are described briefly here in Sections 1.3.2 and 1.3.3 and in detail in the accompanying P2PSx manual [1].

An overview of the **Agentj** software architecture is given in Figure 1.2. At the lower levels we have two distinct environments that **Agentj** application

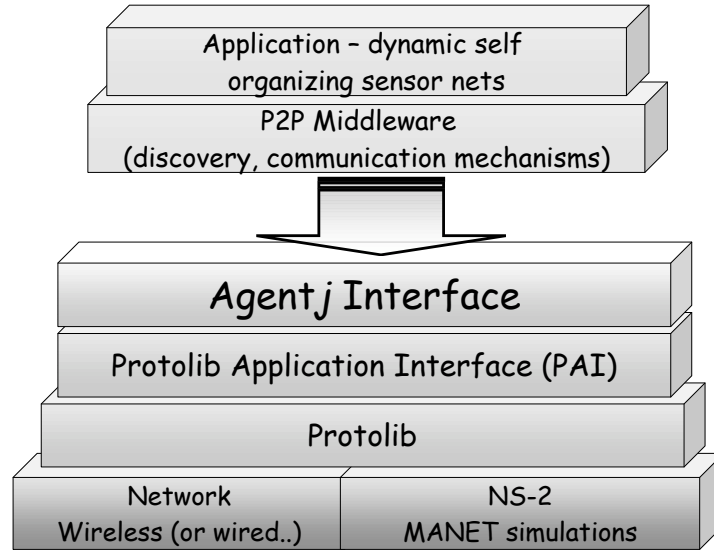


Figure 1.2: An overview of the **Agentj** software stack

can work within: the Internet (or networked environment) or within the NS2 simulation environment. Application typically can switch between the nodes and every effort has been made to make this transition between a simulated environment and a real-world deployment as seamless as possible.

NS-2 [12] is a discrete event simulator that supports the link layer upwards on the OSI stack i.e. the network, transport, session, presentation and application layer, respectively. It can support both wired and wireless simulations and works on most platforms and therefore satisfies the main focus of the project, that is, to test out various P2P discovery and communication mechanisms within various network extremities.

The toolkit that provides the glue for the communication protocols and timing interfaces that can enables **Agentj** to switch between these modes is called Protolib [4]. Protolib implements a switchable communications layer for several communication protocols (e.g. TCP, UDP, Multicast etc) in that the same programming interface can be used to communicate within either environment. To bind to NS2 or a networked environment, typically the application just needs to be recompiled. Protolib is described in detail in Chapter 3.

The PAI interface (described in Chapt. 4) extends the functionality/flexibility of Protolib toolkit by allowing multiple listeners to be attached to sockets or timers. It also provides convenient interfaces for creating and

deleting socket or timing instances and implements an engine for providing the necessary housekeeping. In essence, PAI provides a hosting environment for Protolib to be deployed in Java applications because it fills in the gap between what Java applications expect and what the core Protolib toolkit provides.

Finally, **Agentj** provides a familiar Java interface, based around the *java.net* package, to the underlying functionality provided by the PAI and Protolib interfaces. Specifically **Agentj** implements a Java Native Interface (JNI) binding for the PAI C++ interface and then re-implements this functionality within a set of Java classes that allow Java programmers to use Protolib sockets as if they were native Java ones. The next section describes this interaction in more detail.

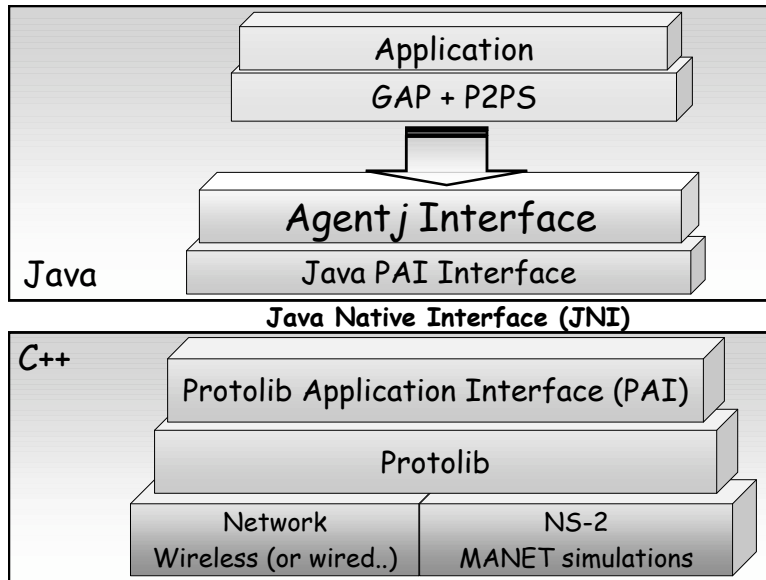


Figure 1.3: The **Agentj** architecture showing the JNI interface, which provides the mapping between the NS2 C++ PAI classes and the Java implementation of the **Agentj** interface and applications.

1.2.1 **Agentj** and the JNI PAI Interface

Since the middleware is written in Java, a JNI bridge is needed in order to map between the C++ NS2 objects and the associated Java objects. This bridging mechanism is required at both the input to the Java application and

at its lower communication layers; that is, first the C++ NS2 agents need to access and attach Java agents to the NS2 agents, then these Java agents need to be able to access the PAI C++ interface in order to pass data between NS2 nodes.

In the first case, the C++ agents create a Java Virtual Machine (JVM) in order to create an environment for running and accessing Java objects. In the second case, we provide a mapping to the PAI and Protolib C++ libraries via the Java PAI interface using JNI. The resulting programming architecture is shown in Figure 1.3.

In this figure, we show the view from an application developer, which illustrates the interface that s/he interacts with. The developer of a Java NS2 applications interacts with a set of high-level Java classes, whilst this functionality is converted to a set of Java PAI function calls, which in turn, is converted via a JNI interface to the C++ PAI interface and down to the Protolib interface. This interaction is described in more detail in Chapter ?? . The next section briefly describes the GAP interface and P2PS Middleware and then we summarize the complete structure in the final section of this chapter.

1.3 The GAP Interface and P2PS

Due to the flexibility needed for comparing different discovery mechanisms the SRSS project have adopted the use of a high-level interface, called the GAP. The GAP is an application-layer interface that has been developed at Cardiff University within the Gridlab [16] and GridOneD projects [17]. The GAP interface provides access to a core set of advertisement, discovery and communication services, which were designed by analysing a number of P2P applications and extracting the core functionality most applications require.

The GAP was motivated by the Grid Application Toolkit (GAT) interface [8], which is committed to conforming to emerging application-level standards that are currently being specified through the SAGA research group at the Global Grid Forum [5]. The GAP provides the asynchronous messaging capabilities for the GAT engine, which is currently undergoing widespread adoption by many application groups worldwide and featured heavily in GRID-START efforts [9].

1.3.1 The GAT

The GAT interface provides a generalised collection of calls to shield Grid applications from implementation details of the underlying Grid middleware, and is being developed in the European GridLab project [16]. The GAT utilises *adaptors* that provide the specific bindings from the GAT interface to the underlying mechanisms that implement this functionality. For example, a *move_file* command may have many GAT adaptors that implement this functionality depending upon the particular execution environment used, such as GridFTP, JXTA pipes or a local *cp* command.

GAT may be referred to as *upperware*, which distinguishes it from middleware (which provides the actual implementation of the underlying functionality). Until recently, application developers typically interact with the middleware directly. However, it is becoming increasingly apparent that this transition from one type of middleware to another is not a trivial one. Using interfaces like GAT, migrating from one middleware environment to another is easier, and typically achieved by setting an environment variable. This is illustrated in the next section where we have implemented an adaptor to bind to P2P middleware for operating in P2P environments as well as the Grid environments supported directly by GridLab. This means that exactly the same Triana implementation can be used within both environments transparently.

1.3.2 The GAP Interface

The Grid Application Prototype Interface (GAP Interface) is a generic application interface providing a subset of the GAT functionality. It is middleware independent, with bindings provided for different Grid middleware such as JXTA and Web Services, as illustrated in Figure 1.4.

Part of the motivation behind the GAP Interface is as a stopgap to enable us to develop distribution mechanisms within Triana while the GridLab GAT is being developed. When the GridLab GAT becomes available the GAT-API will replace the GAP Interface within Triana and should enable Triana to make use of the advanced security, logging and other GridLab services. However, the GAP Interface will live on, both as a simple interface for prototyping Grid and P2P applications, and as an adaptor within the GridLab GAT architecture providing various discovery and communication capabilities. Currently there are three GAP bindings implemented:

JXTA - The original GAP Interface binding was to JXTA [15]. JXTA is a set of protocols for Peer-to-Peer discovery and communication originally developed by Sun Microsystems. Although we achieved some

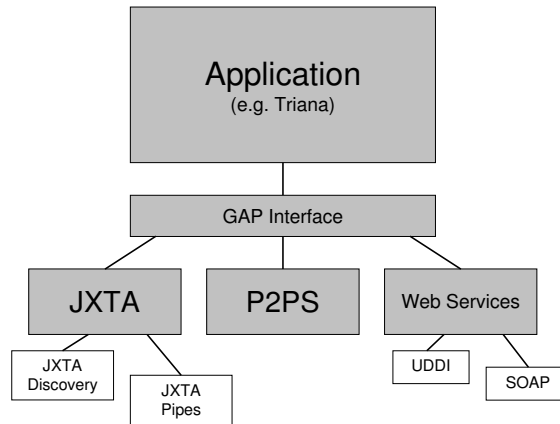


Figure 1.4: The GAP Interface provides a middleware independent interface for developing Grid applications

initial success with JXTA, we have since had problems with the speed and reliability of the JXTA binding.

P2PS - a lightweight Peer-to-Peer middleware. See Section 1.3.3 and for a more detailed description, see the P2PSx manual [1].

Web Services - The most recent GAP binding allows applications to discover and interact with Web Services – using the UDDI registry [24] and the Web Service Invocation Framework (WSIF) [25].

1.3.3 P2PS

P2PS (Peer-to-Peer Simplified) is a lightweight peer-to-peer infrastructure. As the name suggests, P2PS aims to provide a simple collection of middleware that a developer can use to write peer-to-peer style applications, hiding the complexity of other similar architectures such as JXTA [15] and JINI [22].

Briefly, the P2PS infrastructure is based on XML based discovery and communication, which makes it independent of any implementation language and computing hardware. P2PS implementations could exist in any language and there is a specification which can be used to implement such, although at this time we have only built a prototype Java implementation. Furthermore, communication within P2PS is not tied to any single transport protocol, such as TCP/IP, and can be extended to include new protocols, such as Bluetooth

or extend existing ones by writing new endpoint resolvers e.g. we use this approach to write NS2 endpoint resolvers for TCP and UDP.

P2PS has been design to operate in highly dynamic, transient environments and provides an overlay for discovering anything that a peer wants to advertise e.g. specific services, rendezvous (caching) peers, endpoint protocols etc. P2PS dynamically discovers the capabilities of other peers at run-time and can negotiate and match how it communicates and how it organises its peers. This makes P2PS highly suitable for testing out different SRSS discovery mechanisms for two key reasons. First, we can test the discovery mechanisms built into P2PS (Multicast and Unicast) and secondly, we can easily extend this to include other protocols by writing new endpoint resolvers. Thus, we have a core extensible framework for testing and exploring a number of mechanisms both within a simulated environment or within a real-world application.

1.4 Filling in the Gaps

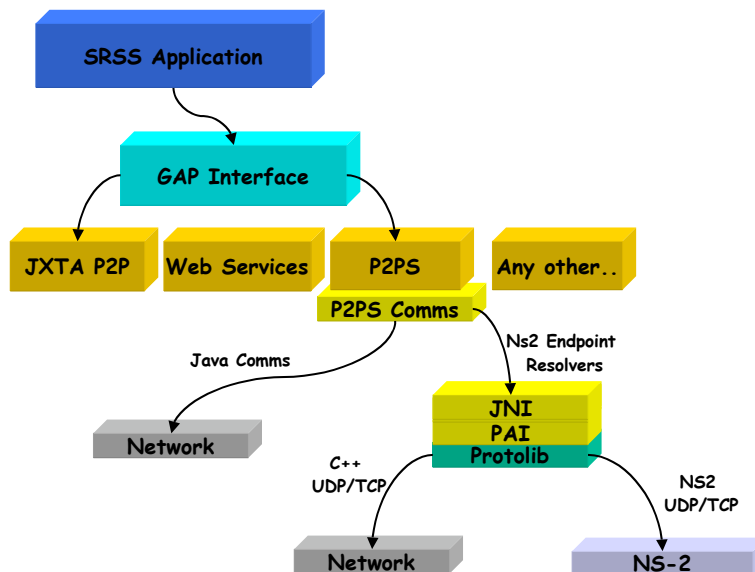


Figure 1.5: An overview of how the SRSS application will use the complete *Agentj* architecture along with the GAP and P2PSx binding.

The resulting architecture is shown in Figure 1.5. The application (i.e.

mobile sensors) can utilise the advertising, discovery and communication mechanisms provided through the GAP, and also hook into any of the underlying GAP bindings.

Currently, we have integrated P2PS into NS2 and also we have implemented a Web services deployment infrastructure using P2PS, allowing standardised Web services to be deployed and tested within this environment also. This means that any pre-written Web service that uses WSDL for its interface and communicates using SOAP can be hosted within NS2 using P2PS.

As illustrated in Figure 1.5, P2PS then uses the *Agentj* Java socket implementation, which in turn, uses the Java PAI interface. The Java PAI interface then maps one-to-one with the C++ PAI interface, which uses Protolib to hook into the lower-level communication layers. This enables the resulting application to be deployed in a networked or an NS-2 simulation environment.

1.5 Conclusion

In this chapter a background and motivation into the **Agentj** project was given. We discussed the project that evolved **Agentj** and how the system fits in with their particular research aims and objectives. We then gave an architectural overview of **Agentj** and discussed the various components and interfaces that make up the entire system. A summary of each component was given, laying out the groundwork for the rest of the chapters in this manual.

Chapter 2

Installing the *Agentj* Toolkit

This chapter describes the installation of core packages needed in order to get the **Agentj** toolkit operating. The core packages needed are:

1. **Protolib**: a core package for adding timers and UDP communication within NS2 [4]
2. **Agentj**: this includes a customized version of the P2PS middleware [20] and the PAI interface to Protolib, described in Chapt. 4 and the JNI interface for attaching Java Objects to NS2 nodes.

2.1 Downloading the Pieces

NS2 version 2.26 is the recommended version for use with **Agentj**. It can be downloaded as a file called *ns-allinone-2.26.tar.gz* from <http://www.isi.edu/nsnam/dist/>.

The source code for **Agentj** and Protolib can be retrieved via CVS from the Protean Forge research site hosted by the United States Naval Research Laboratory at <http://pf.itd.nrl.navy.mil/>. Protolib is listed as its own project on Protean Forge, but **Agentj** is listed within the SRSS project. These project pages contain bug lists, user forums, source code of major file releases, and publically accessible CVS repositories of the latest development source code.

Protolib can be downloaded via CVS by running the following commands:

- `cvs -d :pserver:anonymous@protolib.pf.itd.nrl.navy.mil:/cvsroot/protolib login`
- `cvs -z3 -d :pserver:anonymous@protolib.pf.itd.nrl.navy.mil:/cvsroot/protolib co .`

Agentj and P2PS-x can be retrieved via CVS by running the following commands:

- `cvs -d :pserver:anonymous@srss.pf.itd.nrl.navy.mil:/cvsroot/srss login`
- `cvs -z3 -d :pserver:anonymous@srss.pf.itd.nrl.navy.mil:/cvsroot/srss co agentj`
- `cvs -z3 -d :pserver:anonymous@srss.pf.itd.nrl.navy.mil:/cvsroot/srss co p2ps-x`

The p2ps-x module contains classes required by **Agentj**. After you've successfully checked out these modules, the **Agentj** manual you are probably currently reading will be in `agentj/doc/agentj.pdf`.

2.2 Installing the Protolib NS2 Binding

With the Protolib release there is a supplied Makefile for NS version 2.26 and a README.TXT file in the *ns* directory. The read me file describes the steps involved in installing protolib into NS2. They are as follows:

To use PROTOLIB with ns, you will need to at least modify the ns "Makefile.in" to build the PROTOLIB code into ns. To do this, use the following steps:

- 1) Make a link to the PROTOLIB source directory in the ns source directory. (I use "protolib" for the link name in the steps below).
- 2) Provide paths to the PROTOLIB include files by setting


```
PROTOLIB_INCLUDES = -Iprotolib/common -Iprotolib/ns
```

 and adding `$(PROTOLIB_INCLUDES)` to the "INCLUDES" macro already defined in the ns "Makefile.in"
- 3) Define compile-time CFLAGS needed for the PROTOLIB code by setting


```
PROTOLIB_FLAGS = -DUNIX -DNS2 -DPROTO_DEBUG -DHAVE_ASSERT
```

 and adding `$(PROTOLIB_FLAGS)` to the "CFLAGS" macro already defined in the ns "Makefile.in"

- 4) Add the list of PROTOLIB object files to get compiled and linked during the ns build. For example, set

```
OBJ_PROTOLIB_CPP = \
    protolib/ns/nsProtoAgent.o protolib/common/protoSim.o\
    protolib/common/networkAddress.o \
    protolib/common/protocolTimer.o \
    protolib/common/debug.o
```

and then add `$(OBJ_PROTOLIB_CPP)` to the list in the "OBJ" macro already defined in the ns "Makefile.in"

Note: "nsProtoAgent.cpp" contains a starter ns agent which uses the PROTOLIB ProtocolTimer and UdpSocket classes.

- 5) Add the the rule for .cpp files to ns-2 "Makefile.in":

```
.cpp.o:
@rm -f $@
$(CC) -c $(CFLAGS) $(INCLUDES) -o $@ $.cpp
```

and add to the ns-2 Makefile.in "SRC" macro definition:

```
$(OBJ_CPP:.o=.cpp)
```

- 6) Run `./configure` in the ns source directory to create a new Makefile and then type "make ns" to rebuild ns.

Brian Adamson
<mailto://adamson@itd.nrl.navy.mil>
 18 December 2001

The first thing to take note is that Protolib is basically a plug-in for NS 2 to allowing a trigger mechanism, based on timers and a UDP socket implementation for passing data between NS2 nodes. Therefore, to install this plug-in, you must recompile NS2. It is therefore advisable to build NS2 from scratch then add the protolib plug-in.

2.3 Installing *Agentj*

The *Agentj* toolkit installation follows a similar installation path to Protolib but instead of using a softlink, it uses environment variables within the *Makefile.in* file to point to the installation directory for the source code

for **Agentj**. The installation is provided below and follows a similar style to the Protolib procedure for simplicity. This file can be found in the `src/build/ns2PAIConfig` directory within the **Agentj** source tree.

To install **Agentj** you need to first install Protolib and then modify the ns "Makefile.in" to build the **Agentj** code into ns (there is a Makefile.in file for the ns2.26 release given in the **Agentj** build directory). Before, we describe how to modify the NS2 makefile, let's take a brief look at the environment variables I define that are used by **Agentj** and NS-2.

2.4 Environment Variables

In my `.tcshrc` file, I define the following environment variables, which allow me to specify the installation directories and specify class paths, library paths and NS-2 specifics in one file. The environment variables here are defined in my `.tcshrc` file on my MAC but could easily be converted to `.bat` files (for windows) or `csh` or `bash` shells on Unix systems.

```
setenv AGENTJ /Users/scmijt/Apps/nrl/agentj

setenv AGENTJDEBUG ON

setenv AGENTJXMLCONFIG $AGENTJ/config/AgentJConfig.xml

setenv PATH $NS/bin:/unix:$NS/tcl8.3.2/unix:$NS/tk8.3.2/unix:$PATH

setenv LD_LIBRARY_PATH $AGENTJ/lib/:$NS/otcl-1.0a8:$NS/lib

setenv TCL_LIBRARY $NS/tcl8.3.2/library

setenv CLASSPATH $AGENTJ/classes:$AGENTJ/lib/autolog.jar:
$AGENTJ/lib/log4j-1.2.8.jar
```

For a Linux system, the CLASSPATH should also include paths to `p2ps-x/classes/`, `p2ps-x/lib/`, and the jar files in `p2ps-x/lib/`. As an example, I define the CLASSPATH in my bash environment on a Linux box as follows:

```
export CLASSPATH=/home/iandow/p2ps/development/p2ps-x/classes:\
/home/iandow/p2ps/development/agentj/classes:\
/home/iandow/p2ps/development/agentj/lib/autolog.jar:\
/home/iandow/p2ps/development/agentj/lib/log4j-1.2.8.jar:\
/home/iandow/p2ps/development/p2ps-x/lib/gap.jar:\
/home/iandow/p2ps/development/p2ps-x/lib/jdom.jar:\
/home/iandow/p2ps/development/p2ps-x/lib/p2ps.jar
```

Descriptions of the purpose of these definitions are given as follows:

- **AGENTJ:** is used to specify the installation directory of the agentj package. This is used by the Makefile.in NS-2 makefile and also used within the other environment variables defined here.
- **AGENTJDEBUG:** is used to specify whether you want to turn on logging or not. Within the C++ parts of the code, we use a simple custom logging scheme, whereas within the Java parts of the code, log4j is used. To turn on logging throughout the system, set this environment variable to 'ON'. Any other value (or no definition of this variable) will resort to the default setting i.e. no debugging information will be displayed. Logging is described in more detail in Section 2.7.
- **AGENTJXMLCONFIG:** This environment variable allows you to specify the format for the log4j java logging using a log4j XML file. See the log4j web site [19] for more information on how to specify these. An example configuration is supplied in the *config* directory, called AgentJConfig.xml, as indicated.
- **PATH:** the standard PATH variable for specifying directories that contain executables. Here, I simply include the directories required by NS-2 version 2.26. For more information, see [12]
- **LD_LIBRARY_PATH:** the standard environment variable for specifying where to find libraries. Here, I extend this to include the *Agentjlib* directory plus some directories required by NS2, version 2.26.
- **TCL_LIBRARY:** required for NS2 installation
- **CLASSPATH:** the standard environment variable used to specify the Java classpath. Here, I extend this with the JAR files and directories required by Agentj e.g. agentj classes directory and two JAR files required for the Java logging: autolog.jar and log4j-1.2.8.jar.

2.5 Installation into NS2

To install *Agentj*, use the following steps:

- 1) Install Protolib
- 2) Set the AGENTJ environment variable to point to your installation directory for *Agentj* and create pointers to the various subdirectories for the source in the Makefile.in NS2 file, as follows:

```

AGENTJ_SRC = $(AGENTJ)/src/c
AGENTJ_LIB_DIR = $(AGENTJ)/lib

AGENTJ_C_SRC = $(AGENTJ_SRC)/agentj
AGENTJ_UTILS = $(AGENTJ_SRC)/utils
PAI = $(AGENTJ_SRC)/pai
PAI_IMP = $(PAI)/imp
PAI_API = $(PAI)/api
PAI_AGENT = $(PAI_IMP)/agent
PAI_FACTORY = $(PAI_IMP)/factory
PAI_FACTORY_NET = $(PAI_FACTORY)/net
PAI_FACTORY_NS = $(PAI_FACTORY)/ns
PAI_IMP_JNI = $(PAI_IMP)/jni

```

- 2) Provide paths to the AGENTJ include files by setting

```
AGENTJ_INCLUDES = -I$(JAVA_HOME)/include -I$(AGENTJ_C_SRC) -I$(AGENTJ_UTILS) -I$(PAI) -I$(
```

and adding \$(AGENTJ_INCLUDES) to the "INCLUDES" macro
already defined in the ns "Makefile.in"

On Linux platforms, include the path to jni_md.h in the list of AGENTJ
include files. So, AGENTJ_INCLUDES might look like something like this:

```
AGENTJ_INCLUDES = -I$(JAVA_HOME)/include -I$(JAVA_HOME)/include/linux -I$(AGENTJ_C_SRC) -I$(
```

- 3) Add the list of AGENTJ object files to get compiled and linked
during the ns build. For example, set

```

OBJ_AGENTJ_CPP = $(AGENTJ_UTILS)/LinkedList.o $(PAI_FACTORY)/PAIDispatcher.o \
$(PAI_FACTORY)/PAIEngine.o $(PAI_FACTORY)/PAIFactory.o \
$(PAI_FACTORY)/PAIMultipleListener.o $(PAI_FACTORY)/PAISocket.o \
$(PAI_FACTORY)/PAITimer.o $(PAI_FACTORY)/PAIEnvironment.o \
$(PAI_FACTORY)/PAIListener.o \
$(PAI_FACTORY_NS)/PAINS2UDPSocket.o \
$(PAI_FACTORY_NS)/PAINS2Timer.o \
$(PAI_API)/PAI.o \
$(PAI_AGENT)/PAIAgent.o $(PAI_AGENT)/PAISimpleAgent.o \
$(AGENTJ_C_SRC)/C2JBroker.o $(AGENTJ_C_SRC)/Agentj.o \
$(PAI_IMP_JNI)/JNIBridge.o $(PAI_IMP_JNI)/JNIImp.o

```

and then add \$(OBJ_AGENTJ_CPP) to the list in the "OBJ"
macro already defined in the ns "Makefile.in"

Note: "Agentj.cpp" contains the NS agent for integrating Java objects.

- 4) Add the rule for .cpp files to ns-2 "Makefile.in":

```
.cpp.o:
@rm -f $@
$(CC) -c $(CFLAGS) $(INCLUDES) -o $@ $*.cpp
```

and add to the ns-2 Makefile.in "SRC" macro definition:

```
$(OBJ_CPP:.o=.cpp)
```

(note this has already been done - if you have installed protolib correctly).

5) Create a shared library - define compile-time SHARED Library flags and libraries needed for your platform to create a shared library (this is needed for the JNI binding). On my Mac OS 10.x, these are defined as follows:

```
AGENTJ_LIB = -framework JavaVM
AGENTJ_SHARED_LDFLAGS = -dynamiclib -lresolv
```

and adding \$(AGENTJ_LIB) to the "LIB" macro already defined in the ns "Makefile.in"

and adding a new rule to make the shared library and put it in the correct place:

```
libagentj.jnilib: $(OBJ) common/tclAppInit.o
$(LINK) $(AGENTJ_SHARED_LDFLAGS) -o $@ \
common/tclAppInit.o $(OBJ) $(LIB)
mv libagentj.jnilib $(AGENTJ_LIB_DIR)
```

On a Linux box running a 2.6.7 kernel with version 1.5.0 of Sun's JDK , these flags are defined as follows:

```
AGENTJ_LIB = -L$(JAVA_HOME)/jre/lib/i386/server/ -ljvm
AGENTJ_SHARED_LDFLAGS = -shared
```

Still add \$(AGENTJ_LIB) to the "LIB" macro already defined in the ns "Makefile.in", just as for Macs.

The new rule for making the shared library for Linux should look like this:

```
libagentj.so: $(OBJ) common/tclAppInit.o
$(LINK) $(AGENTJ_SHARED_LDFLAGS) -o $@ common/tclAppInit.o $(OBJ) $(LIB)
mv libagentj.so $(AGENTJ_LIB_DIR)
```

- 6) Run `./configure` in the ns source directory to create a new Makefile
- 7) Type `"make ns"` to rebuild ns - this creates the static library
- 8) Type `"make libagentj.jnilib"` (or `"make libagentj.so"` for Linux platforms) to make the dynamic library needed for the installation of the JNI frameworks.

2.5.1 The NS-2 Makefile for Macintosh

The resulting NS2 Makefile should therefore including both the Protolib and *Agentj* dependencies. A complete version of my Makefile, used to build NS 2 version 2.26 on an Apple Mac, is provided below:

```
# Copyright (c) 1994, 1995, 1996
# The Regents of the University of California. All rights reserved.
#
# Redistribution and use in source and binary forms, with or without
# modification, are permitted provided that: (1) source code distributions
# retain the above copyright notice and this paragraph in its entirety, (2)
# distributions including binary code include the above copyright notice and
# this paragraph in its entirety in the documentation or other materials
# provided with the distribution, and (3) all advertising materials mentioning
# features or use of this software display the following acknowledgement:
# "This product includes software developed by the University of California,
# Lawrence Berkeley Laboratory and its contributors." Neither the name of
# the University nor the names of its contributors may be used to endorse
# or promote products derived from this software without specific prior
# written permission.
# THIS SOFTWARE IS PROVIDED "AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED
# WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF
# MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
#
# @(#) $Header: 2002/10/09 15:34:11

#
# Various configurable paths (remember to edit Makefile.in, not Makefile)
#

# Top level hierarchy
prefix = @prefix@
# Pathname of directory to install the binary
BINDEST = @prefix@/bin
# Pathname of directory to install the man page
MANDEST = @prefix@/man

BLANK = # make a blank space. DO NOT add anything to this line
```

```

# The following will be redefined under Windows (see WIN32 table below)
CC = @CC@
CPP = @CXX@
LINK = $(CPP)
MKDEP = ./conf/mkdep
TCLSH = @V_TCLSH@
TCL2C = @V_TCL2CPP@
AR = ar rc $(BLANK)

RANLIB = @V_RANLIB@
INSTALL = @INSTALL@
LN = ln
TEST = test
RM = rm -f
MV      = mv
PERL = @PERL@

# for diffusion
#DIFF_INCLUDES = "./diffusion3/main ./diffusion3/lib ./diffusion3/nr ./diffusion3/ns"

# Flags for creating a shared library - IANS Additions

CCOPT = @V_CCOPT@
STATIC = @V_STATIC@
LDFLAGS = $(STATIC)
LDOUT = -o $(BLANK)

##### New Protolib Section #####

OBJ_PROTOLIB_CPP = \
    protolib/ns/nsProtoSimAgent.o protolib/common/protoSimAgent.o \
    protolib/ns/nsProtoRouteMgr.o \
    protolib/common/protoSimSocket.o protolib/common/protoAddress.o \
    protolib/common/protoTimer.o protolib/common/protoExample.o \
    protolib/common/protoDebug.o protolib/common/protoRouteMgr.o \
    protolib/common/protoRouteTable.o protolib/common/protoTree.o

##### Protolib Section #####

PROTOLIB = ../../protolib

PROTOLIB_INCLUDES = -I$(PROTOLIB)/common -I$(PROTOLIB)/ns

PROTOLIB_FLAGS = -DNS2 -DSIMULATE -DUNIX -DPROTO_DEBUG -DHAVE_ASSERT -DHAVE_DIRFD

DEFINE = -DTCP_DELAY_BIND_ALL -DNO_TK @V_DEFINE@
@V_DEFINES@ @DEFS@ -DNS_DIFFUSION
-DSMAC_NO_SYNC -DSTL_NAMESPACE=@STL_NAMESPACE@

```

```
-DUSE_SINGLE_ADDRESS_SPACE
```

```
OBJ_PROTOLIB_CPP = \
    $(PROTOLIB)/ns/nsProtoAgent.o $(PROTOLIB)/common/protoSim.o \
    $(PROTOLIB)/common/networkAddress.o $(PROTOLIB)/common/protocolTimer.o \
    $(PROTOLIB)/common/debug.o
```

```
##### AGENTJ Section #####
```

```
# YOU MUST SPECIFY THE AGENTJ environment variable or set this directly here
```

```
AGENTJ_SRC = $(AGENTJ)/src/c
AGENTJ_LIB_DIR = $(AGENTJ)/lib
```

```
AGENTJ_C_SRC = $(AGENTJ_SRC)/agentj
AGENTJ_UTILS = $(AGENTJ_SRC)/utils
PAI = $(AGENTJ_SRC)/pai
PAI_IMP = $(PAI)/imp
PAI_API = $(PAI)/api
PAI_AGENT = $(PAI_IMP)/agent
PAI_FACTORY = $(PAI_IMP)/factory
PAI_FACTORY_NET = $(PAI_FACTORY)/net
PAI_FACTORY_NS = $(PAI_FACTORY)/ns
PAI_IMP_JNI = $(PAI_IMP)/jni
```

```
#Note just include the ns implementation here - NOT the net directory
```

```
AGENTJ_LIB = -framework JavaVM
AGENTJ_SHARED_LDFLAGS = -dynamiclib -lresolv
```

```
AGENTJ_INCLUDES = -I$(JAVA_HOME)/include -I$(AGENTJ_C_SRC) -I$(AGENTJ_UTILS) -I$(PAI) -I$(
```

```
OBJ_AGENTJ_CPP = $(AGENTJ_UTILS)/LinkedList.o $(PAI_FACTORY)/PAIDispatcher.o \
    $(PAI_FACTORY)/PAIEngine.o $(PAI_FACTORY)/PAIFactory.o \
    $(PAI_FACTORY)/PAIMultipleListener.o $(PAI_FACTORY)/PAISocket.o \
    $(PAI_FACTORY)/PAITimer.o $(PAI_FACTORY)/PAIEnvironment.o \
    $(PAI_FACTORY)/PAIListener.o \
    $(PAI_FACTORY_NS)/PAINS2UDPSocket.o \
    $(PAI_FACTORY_NS)/PAINS2Timer.o \
    $(PAI_API)/PAI.o \
    $(PAI_AGENT)/PAIAgent.o $(PAI_AGENT)/PAISimpleAgent.o \
    $(AGENTJ_C_SRC)/C2JBroker.o $(AGENTJ_C_SRC)/Agentj.o \
    $(PAI_IMP_JNI)/JNIBridge.o $(PAI_IMP_JNI)/JNIImp.o
```

```
##### END AGENTJ Section #####
```

```
INCLUDES = \
    $(PROTOLIB_INCLUDES) \
```

```

$(AGENTJ_INCLUDES) \
-I. @V_INCLUDE_X11@ \
@V_INCLUDES@ \
-I./tcp -I./common -I./link -I./queue \
-I./adc -I./apps -I./mac -I./mobile -I./trace \
-I./routing -I./tools -I./classifier -I./mcast \
-I./diffusion3/lib/main -I./diffusion3/lib \
-I./diffusion3/lib/nr -I./diffusion3/ns \
-I./diffusion3/diffusion -I./asim/ -I./qs

LIB = \
@V_LIBS@ \
@V_LIB_X11@ \
@V_LIB@ \
$(AGENTJ_LIB) \
-lm @LIBS@
# -L@libdir@ \

CFLAGS = $(CCOPT) $(DEFINE) $(PROTOLIB_FLAGS) $(AGENTJ_FLAGS)

# Explicitly define compilation rules since SunOS 4's make doesn't like gcc.
# Also, gcc does not remove the .o before forking 'as', which can be a
# problem if you don't own the file but can write to the directory.
.SUFFIXES: .cc # $(.SUFFIXES)

.cc.o:
@rm -f $@
$(CPP) -c $(CFLAGS) $(INCLUDES) -o $@ $.cc

.c.o:
@rm -f $@
$(CC) -c $(CFLAGS) $(INCLUDES) -o $@ *.c

GEN_DIR = gen/
LIB_DIR = lib/
NS = ns
NSX = nsx
NSE = nse

# To allow conf/makefile.win overwrite this macro
# We will set these two macros to empty in conf/makefile.win since VC6.0
# does not seem to support the STL in gcc 2.8 and up.
OBJ_STL = diffusion3/lib/nr/nr.o diffusion3/lib/dr.o \
diffusion3/ns/diffagent.o diffusion3/ns/diffrtg.o \
diffusion3/ns/difftimer.o \
diffusion3/diffusion/diffusion.o \
diffusion3/lib/main/attrs.o \

```

```

diffusion3/lib/main/iodev.o \
diffusion3/lib/main/timers.o \
diffusion3/lib/main/events.o \
diffusion3/lib/main/message.o \
diffusion3/lib/main/stats.o \
diffusion3/lib/main/tools.o \
diffusion3/lib/drivers/rpc_stats.o \
diffusion3/apps/sysfilters/gradient.o \
diffusion3/apps/sysfilters/log.o \
diffusion3/apps/sysfilters/tag.o \
diffusion3/apps/sysfilters/srcrt.o \
diffusion3/lib/diffapp.o \
diffusion3/apps/pingapp/ping_sender.o \
diffusion3/apps/pingapp/ping_receiver.o \
diffusion3/apps/pingapp/ping_common.o \
diffusion3/apps/pingapp/push_receiver.o \
diffusion3/apps/pingapp/push_sender.o \
diffusion3/apps/gear/geo-attr.o \
diffusion3/apps/gear/geo-routing.o \
diffusion3/apps/gear/geo-tools.o \
nix/hdr_nv.o nix/classifier-nix.o \
nix/nixnode.o nix/nixvec.o \
nix/nixroute.o

```

```

NS_TCL_LIB_STL = tcl/lib/ns-diffusion.tcl

```

```

# WIN32: uncomment the following line to include specific make for VC++
# !include <conf/makefile.win>

```

```

OBJ_CC = \
tools/random.o tools/rng.o tools/ranvar.o common/misc.o common/timer-handler.o \
common/scheduler.o common/object.o common/packet.o \
common/ip.o routing/route.o common/connector.o common/ttl.o \
trace/trace.o trace/trace-ip.o \
classifier/classifier.o classifier/classifier-addr.o \
classifier/classifier-hash.o \
classifier/classifier-virtual.o \
classifier/classifier-mcast.o \
classifier/classifier-bst.o \
classifier/classifier-mpath.o mcast/replicator.o \
classifier/classifier-mac.o \
classifier/classifier-qs.o \
classifier/classifier-port.o src_rtg/classifier-sr.o \
src_rtg/sragent.o src_rtg/hdr_src.o adc/ump.o \
qs/qsagent.o qs/hdr_qs.o \
apps/app.o apps/telnet.o tcp/tcplib-telnet.o \
tools/trafgen.o trace/traffictrace.o tools/pareto.o \

```

```

tools/expoo.o tools/cbr_traffic.o \
adc/tbf.o adc/resv.o adc/sa.o tcp/saack.o \
tools/measuremod.o adc/estimator.o adc/adc.o adc/ms-adc.o \
adc/timewindow-est.o adc/acto-adc.o \
    adc/pointsample-est.o adc/salink.o adc/actp-adc.o \
adc/hb-adc.o adc/expavg-est.o\
adc/param-adc.o adc/null-estimator.o \
adc/adaptive-receiver.o apps/vatrcvr.o adc/consrvcvr.o \
common/agent.o common/message.o apps/udp.o \
common/session-rtp.o apps/rtp.o tcp/rtp.o \
common/ivs.o \
tcp/tcp.o tcp/tcp-sink.o tcp/tcp-reno.o \
tcp/tcp-newreno.o \
tcp/tcp-vegas.o tcp/tcp-rbp.o tcp/tcp-full.o tcp/rq.o \
baytcp/tcp-full-bay.o baytcp/ftpc.o baytcp/ftps.o \
tcp/scoreboard.o tcp/scoreboard-rq.o tcp/tcp-sack1.o tcp/tcp-fack.o \
tcp/tcp-asym.o tcp/tcp-asym-sink.o tcp/tcp-fs.o \
tcp/tcp-asym-fs.o tcp/tcp-qs.o \
tcp/tcp-int.o tcp/chost.o tcp/tcp-session.o \
tcp/nilist.o \
tools/integrator.o tools/queue-monitor.o \
tools/flowmon.o tools/loss-monitor.o \
queue/queue.o queue/drop-tail.o \
adc/simple-intserv-sched.o queue/red.o \
queue/semantic-packetqueue.o queue/semantic-red.o \
tcp/ack-recons.o \
queue/sfq.o queue/fq.o queue/drr.o queue/srr.o queue/cbq.o \
queue/jobs.o queue/marker.o queue/demarker.o \
link/hackloss.o queue/errmodel.o queue/fec.o\
link/delay.o tcp/snoop.o \
gaf/gaf.o \
link/dynalink.o routing/rtProtoDV.o common/net-interface.o \
mcast/ctrMcast.o mcast/mcast_ctrl.o mcast/srm.o \
common/sessionhelper.o queue/delaymodel.o \
mcast/srm-ssm.o mcast/srm-topo.o \
apps/mftp.o apps/mftp_snd.o apps/mftp_rcv.o \
apps/codeword.o \
routing/alloc-address.o routing/address.o \
$(LIB_DIR)int.Vec.o $(LIB_DIR)int.RVec.o \
$(LIB_DIR)dmalloc_support.o \
webcache/http.o webcache/tcp-simple.o webcache/pagepool.o \
webcache/invalid-agent.o webcache/tcpapp.o webcache/http-aux.o \
webcache/mcache.o webcache/webtraf.o \
webcache/webserver.o \
webcache/logweb.o \
empweb/empweb.o \
empweb/empftp.o \
realaudio/realaudio.o \
mac/lanRouter.o classifier/filter.o \

```

```

common/pkt-counter.o \
common/Decapsulator.o common/Encapsulator.o \
common/encap.o \
mac/channel.o mac/mac.o mac/ll.o mac/mac-802_11.o \
mac/mac-802_3.o mac/mac-tdma.o mac/smac.o \
mobile/mip.o mobile/mip-reg.o mobile/gridkeeper.o \
mobile/propagation.o mobile/tworayground.o \
mobile/antenna.o mobile/omni-antenna.o \
mobile/shadowing.o mobile/shadowing-vis.o mobile/dumb-agent.o \
common/bi-connector.o common/node.o \
common/mobilenode.o \
mac/arp.o mobile/god.o mobile/dem.o \
mobile/topography.o mobile/modulation.o \
queue/priqueue.o queue/dsr-priqueue.o \
mac/phy.o mac/wired-phy.o mac/wireless-phy.o \
mac/mac-timers.o trace/cmu-trace.o mac/varp.o \
dsdv/dsdv.o dsdv/rtable.o queue/rtqueue.o \
routing/rtable.o \
imep/imep.o imep/dest_queue.o imep/imep_api.o \
imep/imep_rt.o imep/rxmit_queue.o imep/imep_timers.o \
imep/imep_util.o imep/imep_io.o \
tora/tora.o tora/tora_api.o tora/tora_dest.o \
tora/tora_io.o tora/tora_logs.o tora/tora_neighbor.o \
dsr/dsragent.o dsr/hdr_sr.o dsr/mobicache.o dsr/path.o \
dsr/requesttable.o dsr/routecache.o dsr/add_sr.o \
dsr/dsr_proto.o dsr/flowstruct.o dsr/linkcache.o \
dsr/simplecache.o dsr/sr_forwarder.o \
aodv/aodv_logs.o aodv/aodv.o \
aodv/aodv_rtable.o aodv/aodv_rqueue.o \
common/ns-process.o \
satellite/satgeometry.o satellite/sathandoff.o \
satellite/satlink.o satellite/satnode.o \
satellite/satposition.o satellite/satroute.o \
satellite/sattrace.o \
rap/raplist.o rap/rap.o rap/media-app.o rap/utilities.o \
common/fsm.o tcp/tcp-abs.o \
diffusion/diffusion.o diffusion/diff_rate.o diffusion/diff_prob.o \
diffusion/diff_sink.o diffusion/flooding.o diffusion/omni_mcast.o \
diffusion/hash_table.o diffusion/routing_table.o diffusion/iflist.o \
tcp/tfrc.o tcp/tfrc-sink.o mobile/energy-model.o apps/ping.o tcp/tcp-rfc793edu.o \
queue/rio.o queue/semantic-rio.o tcp/tcp-sack-rh.o tcp/scoreboard-rh.o \
plm/loss-monitor-plm.o plm/cbr-traffic-PP.o \
linkstate/hdr-ls.o \
mpls/classifier-addr-mpls.o mpls/ldp.o mpls/mpls-module.o \
routing/rtable.o classifier/classifier-hier.o \
routing/addr-params.o \
routealgo/rnode.o \
routealgo/bfs.o \
routealgo/rbitmap.o \

```

```

routealgo/rlookup.o \
routealgo/routealgo.o \
diffserv/dsred.o diffserv/dsredq.o \
diffserv/dsEdge.o diffserv/dsCore.o \
diffserv/dsPolicy.o diffserv/ew.o \
queue/red-pd.o queue/pi.o queue/vq.o queue/rem.o \
queue/gk.o \
pushback/rate-limit.o pushback/rate-limit-strategy.o \
pushback/ident-tree.o pushback/agg-spec.o \
pushback/logging-data-struct.o \
pushback/rate-estimator.o \
pushback/pushback-queue.o pushback/pushback.o \
common/parentnode.o trace/basetrace.o \
common/simulator.o asim/asim.o \
common/scheduler-map.o common/splay-scheduler.o \
linkstate/ls.o linkstate/rtProtoLS.o \
pgm/classifier-pgm.o pgm/pgm-agent.o pgm/pgm-sender.o \
pgm/pgm-receiver.o mcast/rcvbuf.o \
mcast/classifier-lms.o mcast/lms-agent.o mcast/lms-receiver.o \
mcast/lms-sender.o \
@V_STL_OBJ@

# don't allow comments to follow continuation lines

# mac-csma.o mac-multihop.o \
# sensor-nets/landmark.o mac-simple-wireless.o \
# sensor-nets/tags.o sensor-nets/sensor-query.o \
# sensor-nets/flood-agent.o \

# what was here before is now in emulate/
OBJ_C =

OBJ_COMPAT = $(OBJ_GETOPT) common/win32.o
#XXX compat/win32x.o compat/tkConsole.o

OBJ_EMULATE_CC = \
emulate/net-ip.o \
emulate/net.o \
emulate/tap.o \
emulate/ether.o \
emulate/internet.o \
emulate/ping_responder.o \
emulate/arp.o \
emulate/icmp.o \
emulate/net-pcap.o \
emulate/nat.o \
emulate/iptables.o \
emulate/tcptap.o

```

```

OBJ_EMULATE_C = \
emulate/inet.o

OBJ_GEN = $(GEN_DIR)version.o $(GEN_DIR)ns_tcl.o $(GEN_DIR)ptypes.o

SRC = $(OBJ_C:.o=.c) $(OBJ_CC:.o=.cc) \
$(OBJ_EMULATE_C:.o=.c) $(OBJ_EMULATE_CC:.o=.cc) \
$(OBJ_CPP:.o=.cpp) \
common/tclAppInit.cc common/tkAppInit.cc

OBJ = $(OBJ_C) $(OBJ_CC) $(OBJ_GEN) $(OBJ_COMPAT) $(OBJ_PROTOLIB_CPP) $(OBJ_AGENTJ_CPP)

CLEANFILES = ns nse nsx ns.dyn $(OBJ) $(OBJ_EMULATE_CC) \
$(OBJ_EMULATE_C) common/tclAppInit.o \
$(GEN_DIR)* $(NS).core core core.$(NS) core.$(NSX) core.$(NSE) \
common/ptypes2tcl common/ptypes2tcl.o

SUBDIRS=\
indep-utils/cmu-scen-gen/setdest \
indep-utils/webtrace-conv/dec \
indep-utils/webtrace-conv/epa \
indep-utils/webtrace-conv/nlanr \
indep-utils/webtrace-conv/ucb

BUILD_NSE = @build_nse@

all: $(NS) $(BUILD_NSE) all-recursive

all-recursive:
for i in $(SUBDIRS); do ( cd $$i; $(MAKE) all; ) done

$(NS): $(OBJ) common/tclAppInit.o Makefile
$(LINK) $(LDFLAGS) $(LDOUT)$@ \
common/tclAppInit.o $(OBJ) $(LIB)

Makefile: Makefile.in
@echo "Makefile.in is newer than Makefile."
@echo "You need to re-run configure."
false

$(NSE): $(OBJ) common/tclAppInit.o $(OBJ_EMULATE_CC) $(OBJ_EMULATE_C)
$(LINK) $(LDFLAGS) $(LDOUT)$@ \
common/tclAppInit.o $(OBJ) \
$(OBJ_EMULATE_CC) $(OBJ_EMULATE_C) $(LIB)

ns.dyn: $(OBJ) common/tclAppInit.o
$(LINK) $(LDFLAGS) -o $@ \

```

```

common/tclAppInit.o $(OBJ) $(LIB)

libagentj.jnilib: $(OBJ) common/tclAppInit.o
$(LINK) $(AGENTJ_SHARED_LDFLAGS) -o $$ \
common/tclAppInit.o $(OBJ) $(LIB)
mv libagentj.jnilib $(AGENTJ_LIB_DIR)

PURIFY = purify -cache-dir=/tmp
ns-pure: $(OBJ) common/tclAppInit.o
$(PURIFY) $(LINK) $(LDFLAGS) -o $$ \
common/tclAppInit.o $(OBJ) $(LIB)

NS_TCL_LIB = \
tcl/lib/ns-compat.tcl \
tcl/lib/ns-default.tcl \
tcl/lib/ns-errmodel.tcl \
tcl/lib/ns-lib.tcl \
tcl/lib/ns-link.tcl \
tcl/lib/ns-mobilenode.tcl \
tcl/lib/ns-sat.tcl \
tcl/lib/ns-cmutrace.tcl \
tcl/lib/ns-node.tcl \
tcl/lib/ns-rtmodule.tcl \
tcl/lib/ns-hiernode.tcl \
tcl/lib/ns-packet.tcl \
tcl/lib/ns-queue.tcl \
tcl/lib/ns-source.tcl \
tcl/lib/ns-nam.tcl \
tcl/lib/ns-trace.tcl \
tcl/lib/ns-agent.tcl \
tcl/lib/ns-random.tcl \
tcl/lib/ns-namsupp.tcl \
tcl/lib/ns-address.tcl \
tcl/lib/ns-intserv.tcl \
tcl/lib/ns-autoconf.tcl \
tcl/rtp/session-rtp.tcl \
tcl/lib/ns-mip.tcl \
tcl/rtglib/dynamics.tcl \
tcl/rtglib/route-proto.tcl \
tcl/rtglib/algo-route-proto.tcl \
tcl/rtglib/ns-rtProtoLS.tcl \
    tcl/interface/ns-iface.tcl \
tcl/mcast/BST.tcl \
    tcl/mcast/ns-mcast.tcl \
    tcl/mcast/McastProto.tcl \
    tcl/mcast/DM.tcl \
tcl/mcast/srm.tcl \
tcl/mcast/srm-adaptive.tcl \
tcl/mcast/srm-ssm.tcl \

```

```

tcl/mcast/timer.tcl \
tcl/mcast/McastMonitor.tcl \
tcl/mcast/mftp_snd.tcl \
tcl/mcast/mftp_rcv.tcl \
tcl/mcast/mftp_rcv_stat.tcl \
tcl/mobility/dsdv.tcl \
tcl/mobility/dsr.tcl \
    tcl/ctr-mcast/CtrMcast.tcl \
    tcl/ctr-mcast/CtrMcastComp.tcl \
    tcl/ctr-mcast/CtrRPCComp.tcl \
tcl/rlm/rlm.tcl \
tcl/rlm/rlm-ns.tcl \
tcl/session/session.tcl \
tcl/lib/ns-route.tcl \
tcl/emulate/ns-emulate.tcl \
tcl/lan/vlan.tcl \
tcl/lan/abslan.tcl \
tcl/lan/ns-ll.tcl \
tcl/lan/ns-mac.tcl \
tcl/webcache/http-agent.tcl \
tcl/webcache/http-server.tcl \
tcl/webcache/http-cache.tcl \
tcl/webcache/http-mcache.tcl \
tcl/webcache/webtraf.tcl \
tcl/webcache/empweb.tcl \
tcl/webcache/empftp.tcl \
tcl/plm/plm.tcl \
tcl/plm/plm-ns.tcl \
tcl/plm/plm-topo.tcl \
tcl/mpls/ns-mpls-classifier.tcl \
tcl/mpls/ns-mpls-ldpagent.tcl \
tcl/mpls/ns-mpls-node.tcl \
tcl/mpls/ns-mpls-simulator.tcl \
tcl/lib/ns-pushback.tcl \
tcl/lib/ns-srcrt.tcl \
tcl/mcast/ns-lms.tcl \
tcl/lib/ns-qsnode.tcl \
@V_NS_TCL_LIB_STL@

```

```
$(GEN_DIR)ns_tcl.cc: $(NS_TCL_LIB)
```

```
$(TCLSH) bin/tcl-expand.tcl tcl/lib/ns-lib.tcl @V_NS_TCL_LIB_STL@ | $(TCL2C) et_ns_lib > $
```

```
$(GEN_DIR)version.c: VERSION
```

```
$(RM) $@
```

```
$(TCLSH) bin/string2c.tcl version_string < VERSION > $@
```

```
$(GEN_DIR)ptypes.cc: common/ptypes2tcl common/packet.h
```

```
./common/ptypes2tcl > $@
```

```

common/ptypes2tcl: common/ptypes2tcl.o
$(LINK) $(LDFLAGS) $(LDOUT)$@ common/ptypes2tcl.o

common/ptypes2tcl.o: common/ptypes2tcl.cc common/packet.h

install: force install-ns install-man install-recursive

install-ns: force
$(INSTALL) -m 555 -o bin -g bin ns $(DESTDIR)$(BINDEST)

install-man: force
$(INSTALL) -m 444 -o bin -g bin ns.1 $(DESTDIR)$(MANDEST)/man1

install-recursive: force
for i in $(SUBDIRS); do ( cd $$i; $(MAKE) install; ) done

clean:
$(RM) $(CLEANFILES)

AUTOCONF_GEN = tcl/lib/ns-autoconf.tcl
distclean: distclean-recursive
$(RM) $(CLEANFILES) Makefile config.cache config.log config.status \
    autoconf.h gnuc.h os-proto.h $(AUTOCONF_GEN); \
$(MV) .configure .configure- ;\
echo "Moved .configure to .configure-"

distclean-recursive:
for i in $(SUBDIRS); do ( cd $$i; $(MAKE) clean; $(RM) Makefile; ) done

tags: force
ctags -wtd *.cc *.h webcache/*.cc webcache/*.h dsdv/*.cc dsdv/*.h \
dsr/*.cc dsr/*.h webcache/*.cc webcache/*.h lib/*.cc lib/*.h \
../Tcl/*.cc ../Tcl/*.h

TAGS: force
etags *.cc *.h webcache/*.cc webcache/*.h dsdv/*.cc dsdv/*.h \
dsr/*.cc dsr/*.h webcache/*.cc webcache/*.h lib/*.cc lib/*.h \
../Tcl/*.cc ../Tcl/*.h

tcl/lib/TAGS: force
( \
cd tcl/lib; \
$(TCLSH) ../../bin/tcl-expand.tcl ns-lib.tcl | grep '^### tcl-expand.tcl:
begin' | awk '{print $$5}' >.tcl_files; \
etags --lang=none -r '/^[ \t]*proc[ \t]+\([^ \t]+\)/\1/' 'cat .tcl_files'; \
etags --append --lang=none -r '/^\([A-Z][^ \t]+\)[ \t]+\(instproc|proc\)[
 \t]+\([^ \t]+\)[ \t]+\1::\3/' 'cat .tcl_files'; \
)

```

```

depend: $(SRC)
$(MKDEP) $(CFLAGS) $(INCLUDES) $(SRC)

srctar:
@cwd='pwd' ; dir='basename $$cwd' ; \
  name=ns-'cat VERSION | tr A-Z a-z' ; \
  tar=ns-src-'cat VERSION'.tar.gz ; \
  list="" ; \
  for i in 'cat FILES' ; do list="$$list $$name/$$i" ; done; \
  echo \
  "(rm -f $$tar; cd .. ; ln -s $$dir $$name)" ; \
  (rm -f $$tar; cd .. ; ln -s $$dir $$name) ; \
  echo \
  "(cd .. ; tar cfh $$tar [lots of files])" ; \
  (cd .. ; tar cfh - $$list) | gzip -c > $$tar ; \
  echo \
  "rm ../$$name; chmod 444 $$tar" ; \
  rm ../$$name; chmod 444 $$tar

force:

test: force
./validate

.cpp.o:
@rm -f $$@
$(CC) -c $(CFLAGS) $(INCLUDES) -o $$@ $.cpp

# Create makefile.vc for Win32 development by replacing:
# "# !include ..." -> "!include ..."
makefile.vc: Makefile.in
$(PERL) bin/gen-vcmake.pl < Makefile.in > makefile.vc
# $(PERL) -pe 's/^# (\!include)/\!include/o' < Makefile.in > makefile.vc

```

2.5.2 The NS-2 Makefile for Linux

At the beginning of section 2.5, instructions were given for including Protolib and *Agentj* dependencies in the NS-2 Makefile for Linux platforms. A complete version of my Makefile, used to build NS-2 version 2.26 on a Linux box running a 2.6.7 kernel and version 1.5.0 of Sun's JDK, is provided below:

```

# Generated automatically from Makefile.in by configure.
# Copyright (c) 1994, 1995, 1996
# The Regents of the University of California. All rights reserved.
#
# Redistribution and use in source and binary forms, with or without
# modification, are permitted provided that: (1) source code distributions

```

```

# retain the above copyright notice and this paragraph in its entirety, (2)
# distributions including binary code include the above copyright notice and
# this paragraph in its entirety in the documentation or other materials
# provided with the distribution, and (3) all advertising materials mentioning
# features or use of this software display the following acknowledgement:
# 'This product includes software developed by the University of California,
# Lawrence Berkeley Laboratory and its contributors.' Neither the name of
# the University nor the names of its contributors may be used to endorse
# or promote products derived from this software without specific prior
# written permission.
# THIS SOFTWARE IS PROVIDED 'AS IS' AND WITHOUT ANY EXPRESS OR IMPLIED
# WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF
# MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
#
# @(#) $Header: 2002/10/09 15:34:11

#
# Various configurable paths (remember to edit Makefile.in, not Makefile)
#

# Top level hierarchy
prefix = /usr/local
# Pathname of directory to install the binary
BINDEST = /usr/local/bin
# Pathname of directory to install the man page
MANDEST = /usr/local/man

BLANK = # make a blank space. DO NOT add anything to this line

# The following will be redefined under Windows (see WIN32 table below)
#CC = gcc
#CPP = c++
CC = gcc
CPP = g++
LINK = $(CPP)
MKDEP = ./conf/mkdep
TCLSH = /home/iandow/netsim/p2ps-ns2/ns-allinone-2.26/bin/tclsh8.3
TCL2C = ../tclcl-1.0b13/tcl2c++
AR = ar rc $(BLANK)

RANLIB = ranlib
INSTALL = /usr/bin/install -c
LN = ln
TEST = test
RM = rm -f
MV = mv
PERL = /usr/bin/perl

# for diffusion

```

```
#DIFF_INCLUDES = "./diffusion3/main ./diffusion3/lib ./diffusion3/nr ./diffusion3/ns"

CCOPT =
STATIC =
LDFLAGS = $(STATIC) -Wl,--rpath -Wl,$(JAVA_HOME)/jre/lib/i386/server/$(JAVA_HOME)/jre/lib
LDOUT = -o $(BLANK)

##### Protolib Section #####

PROTOLIB = ./protolib
PROTOLIB_INCLUDES = -I$(PROTOLIB)/common -I$(PROTOLIB)/ns
PROTOLIB_FLAGS = -DUNIX -DNS2 -DPROTO_DEBUG -DHAVE_ASSERT

OBJ_PROTOLIB_CPP = protolib/ns/nsProtoAgent.o \
protolib/common/protoSim.o \
protolib/common/networkAddress.o \
protolib/common/protocolTimer.o \
protolib/common/debug.o

OBJ_AGENTJ_CPP = $(AGENTJ_UTILS)/LinkedList.o $(PAI_FACTORY)/PAIDispatcher.o \
$(PAI_FACTORY)/PAIEngine.o $(PAI_FACTORY)/PAIFactory.o \
$(PAI_FACTORY)/PAIMultipleListener.o $(PAI_FACTORY)/PAISocket.o \
$(PAI_FACTORY)/PAITimer.o $(PAI_FACTORY)/PAIEnvironment.o \
$(PAI_FACTORY)/PAIListener.o \
$(PAI_FACTORY_NS)/PAINS2UDPSocket.o \
$(PAI_FACTORY_NS)/PAINS2Timer.o \
$(PAI_API)/PAI.o \
$(PAI_AGENT)/PAIAgent.o $(PAI_AGENT)/PAISimpleAgent.o \
$(AGENTJ_C_SRC)/C2JBroker.o $(AGENTJ_C_SRC)/Agentj.o \
$(PAI_IMP_JNI)/JNIBridge.o $(PAI_IMP_JNI)/JNIImp.o

##### AgentJ Section #####

AGENTJ_SRC = $(AGENTJ)/src/c
AGENTJ_LIB_DIR = $(AGENTJ)/lib
AGENTJ_C_SRC = $(AGENTJ_SRC)/agentj
AGENTJ_UTILS = $(AGENTJ_SRC)/utils
PAI = $(AGENTJ_SRC)/pai
PAI_IMP = $(PAI)/imp
PAI_API = $(PAI)/api
PAI_AGENT = $(PAI_IMP)/agent
PAI_FACTORY = $(PAI_IMP)/factory
PAI_FACTORY_NET = $(PAI_FACTORY)/net
PAI_FACTORY_NS = $(PAI_FACTORY)/ns
PAI_IMP_JNI = $(PAI_IMP)/jni

AGENTJ_INCLUDES = -I$(JAVA_HOME)/include -I$(JAVA_HOME)/include/linux -I$(AGENTJ_C_SRC) -I
```

```

AGENTJ_LIB = -L$(JAVA_HOME)/jre/lib/i386/server/ -ljvm
AGENTJ_SHARED_LDFLAGS = -shared

DEFINE = -DTCP_DELAY_BIND_ALL -DNO_TK -DTCLCL_CLASSINSTVAR -DNDEBUG -DLINUX_TCP_HEADER -DUSE_SHM

INCLUDES = \
$(PROTOLIB_INCLUDES) \
$(AGENTJ_INCLUDES) \
-I. \
-I/home/iandow/netnsim/p2ps-ns2/ns-allinone-2.26/tclcl-1.0b13 -I/home/iandow/netnsim/p2ps-ns2/ns-all
-I/home/iandow/netnsim/p2ps-ns2/ns-allinone-2.26/ns-2.26/common \
-I/home/iandow/netnsim/p2ps-ns2/ns-allinone-2.26/tclcl-1.0b13 \
-I/home/iandow/netnsim/p2ps-ns2/ns-allinone-2.26/otcl-1.0a8 \
-I/home/iandow/netnsim/p2ps-ns2/ns-allinone-2.26/include \
-I/home/iandow/netnsim/p2ps-ns2/ns-allinone-2.26/include \
-I/usr/include/pcap \
-I./tcp -I./common -I./link -I./queue \
-I./adc -I./apps -I./mac -I./mobile -I./trace \
-I./routing -I./tools -I./classifier -I./mcast \
-I./diffusion3/lib/main -I./diffusion3/lib \
-I./diffusion3/lib/nr -I./diffusion3/ns \
-I./diffusion3/diffusion -I./asim/ -I./qs

LIB = \
-L/home/iandow/netnsim/p2ps-ns2/ns-allinone-2.26/tclcl-1.0b13 -ltclcl -L/home/iandow/netnsim/p2ps-ns
-L/usr/X11R6/lib -lXext -lX11 \
-lns1 -lpcap -ldl \
$(AGENTJ_LIB) \
-lm
# -L${exec_prefix}/lib \

CFLAGS = -g $(CCOPT) $(DEFINE) $(PROTOLIB_FLAGS)

# Explicitly define compilation rules since SunOS 4's make doesn't like gcc.
# Also, gcc does not remove the .o before forking 'as', which can be a
# problem if you don't own the file but can write to the directory.
.SUFFIXES: .cc # $(.SUFFIXES)

.cc.o:
@rm -f $@
$(CPP) -c $(CFLAGS) $(INCLUDES) -o $@ $.cc

.c.o:
@rm -f $@
$(CC) -c $(CFLAGS) $(INCLUDES) -o $@ $.c

.cpp.o: @rm -f $@
$(CC) -c $(CFLAGS) $(INCLUDES) -o $@ $.cpp

```

```

GEN_DIR = gen/
LIB_DIR = lib/
NS = ns
NSX = nsx
NSE = nse

# To allow conf/makefile.win overwrite this macro
# We will set these two macros to empty in conf/makefile.win since VC6.0
# does not seem to support the STL in gcc 2.8 and up.
OBJ_STL = diffusion3/lib/nr/nr.o diffusion3/lib/dr.o \
diffusion3/ns/diffagent.o diffusion3/ns/diffrtg.o \
diffusion3/ns/difftimer.o \
diffusion3/diffusion/diffusion.o \
diffusion3/lib/main/attrs.o \
diffusion3/lib/main/iodev.o \
diffusion3/lib/main/timers.o \
diffusion3/lib/main/events.o \
diffusion3/lib/main/message.o \
diffusion3/lib/main/stats.o \
diffusion3/lib/main/tools.o \
diffusion3/lib/drivers/rpc_stats.o \
diffusion3/apps/sysfilters/gradient.o \
diffusion3/apps/sysfilters/log.o \
diffusion3/apps/sysfilters/tag.o \
diffusion3/apps/sysfilters/srcrt.o \
diffusion3/lib/diffapp.o \
diffusion3/apps/pingapp/ping_sender.o \
diffusion3/apps/pingapp/ping_receiver.o \
diffusion3/apps/pingapp/ping_common.o \
diffusion3/apps/pingapp/push_receiver.o \
diffusion3/apps/pingapp/push_sender.o \
diffusion3/apps/gear/geo-attr.o \
diffusion3/apps/gear/geo-routing.o \
diffusion3/apps/gear/geo-tools.o \
nix/hdr_nv.o nix/classifier-nix.o \
nix/nixnode.o nix/nixvec.o \
nix/nixroute.o

NS_TCL_LIB_STL = tcl/lib/ns-diffusion.tcl

# WIN32: uncomment the following line to include specific make for VC++
# !include <conf/makefile.win>

OBJ_CC = \
tools/random.o tools/rng.o tools/ranvar.o common/misc.o common/timer-handler.o \
common/scheduler.o common/object.o common/packet.o \
common/ip.o routing/route.o common/connector.o common/ttl.o \

```

```

trace/trace.o trace/trace-ip.o \
classifier/classifier.o classifier/classifier-addr.o \
classifier/classifier-hash.o \
classifier/classifier-virtual.o \
classifier/classifier-mcast.o \
classifier/classifier-bst.o \
classifier/classifier-mpath.o mcast/replicator.o \
classifier/classifier-mac.o \
classifier/classifier-qs.o \
classifier/classifier-port.o src_rtg/classifier-sr.o \
    src_rtg/sragent.o src_rtg/hdr_src.o adc/ump.o \
qs/qsagent.o qs/hdr_qs.o \
apps/app.o apps/telnet.o tcp/tcplib-telnet.o \
tools/trafgen.o trace/traffictrace.o tools/pareto.o \
tools/expoo.o tools/cbr_traffic.o \
adc/tbf.o adc/resv.o adc/sa.o tcp/saack.o \
tools/measuremod.o adc/estimator.o adc/adc.o adc/ms-adc.o \
adc/timewindow-est.o adc/acto-adc.o \
    adc/pointsample-est.o adc/salink.o adc/actp-adc.o \
adc/hb-adc.o adc/expavg-est.o \
adc/param-adc.o adc/null-estimator.o \
adc/adaptive-receiver.o apps/vatrcvr.o adc/consrvcvr.o \
common/agent.o common/message.o apps/udp.o \
common/session-rtp.o apps/rtp.o tcp/rtp.o \
common/ivs.o \
tcp/tcp.o tcp/tcp-sink.o tcp/tcp-reno.o \
tcp/tcp-newreno.o \
tcp/tcp-vegas.o tcp/tcp-rbp.o tcp/tcp-full.o tcp/rq.o \
baytcp/tcp-full-bay.o baytcp/ftpc.o baytcp/ftps.o \
tcp/scoreboard.o tcp/scoreboard-rq.o tcp/tcp-sack1.o tcp/tcp-fack.o \
tcp/tcp-asym.o tcp/tcp-asym-sink.o tcp/tcp-fs.o \
tcp/tcp-asym-fs.o tcp/tcp-qs.o \
tcp/tcp-int.o tcp/chost.o tcp/tcp-session.o \
tcp/nilist.o \
tools/integrator.o tools/queue-monitor.o \
tools/flowmon.o tools/loss-monitor.o \
queue/queue.o queue/drop-tail.o \
adc/simple-intserv-sched.o queue/red.o \
queue/semantic-packetqueue.o queue/semantic-red.o \
tcp/ack-recons.o \
queue/sfq.o queue/fq.o queue/drr.o queue/srr.o queue/cbq.o \
queue/jobs.o queue/marker.o queue/demarker.o \
link/hackloss.o queue/errmodel.o queue/fec.o \
link/delay.o tcp/snoop.o \
gaf/gaf.o \
link/dynalink.o routing/rtpProtoDV.o common/net-interface.o \
mcast/ctrMcast.o mcast/mcast_ctrl.o mcast/srm.o \
common/sessionhelper.o queue/delaymodel.o \
mcast/srm-ssm.o mcast/srm-topo.o \

```

```

apps/mftp.o apps/mftp_snd.o apps/mftp_rcv.o \
apps/codeword.o \
routing/alloc-address.o routing/address.o \
$(LIB_DIR)int.Vec.o $(LIB_DIR)int.RVec.o \
$(LIB_DIR)dmalloc_support.o \
webcache/http.o webcache/tcp-simple.o webcache/pagepool.o \
webcache/invalid-agent.o webcache/tcpapp.o webcache/http-aux.o \
webcache/mcache.o webcache/webtraf.o \
webcache/webserver.o \
webcache/logweb.o \
empweb/empweb.o \
empweb/empftp.o \
realaudio/realaudio.o \
mac/lanRouter.o classifier/filter.o \
common/pkt-counter.o \
common/Decapsulator.o common/Encapsulator.o \
common/encap.o \
mac/channel.o mac/mac.o mac/ll.o mac/mac-802_11.o \
mac/mac-802_3.o mac/mac-tdma.o mac/smac.o \
mobile/mip.o mobile/mip-reg.o mobile/gridkeeper.o \
mobile/propagation.o mobile/tworayground.o \
mobile/antenna.o mobile/omni-antenna.o \
mobile/shadowing.o mobile/shadowing-vis.o mobile/dumb-agent.o \
common/bi-connector.o common/node.o \
common/mobilenode.o \
mac/arp.o mobile/god.o mobile/dem.o \
mobile/topography.o mobile/modulation.o \
queue/priqueue.o queue/dsr-priqueue.o \
mac/phy.o mac/wired-phy.o mac/wireless-phy.o \
mac/mac-timers.o trace/cmu-trace.o mac/varp.o \
dsdv/dsdv.o dsdv/rtable.o queue/rtqueue.o \
routing/rtable.o \
imep/imep.o imep/dest_queue.o imep/imep_api.o \
imep/imep_rt.o imep/rxmit_queue.o imep/imep_timers.o \
imep/imep_util.o imep/imep_io.o \
tora/tora.o tora/tora_api.o tora/tora_dest.o \
tora/tora_io.o tora/tora_logs.o tora/tora_neighbor.o \
dsr/dsragent.o dsr/hdr_sr.o dsr/mobicache.o dsr/path.o \
dsr/requesttable.o dsr/routecache.o dsr/add_sr.o \
dsr/dsr_proto.o dsr/flowstruct.o dsr/linkcache.o \
dsr/simplecache.o dsr/sr_forwarder.o \
aodv/aodv_logs.o aodv/aodv.o \
aodv/aodv_rtable.o aodv/aodv_rqueue.o \
common/ns-process.o \
satellite/satgeometry.o satellite/sathandoff.o \
satellite/satlink.o satellite/satnode.o \
satellite/satposition.o satellite/satroute.o \
satellite/sattrace.o \
rap/raplist.o rap/rap.o rap/media-app.o rap/utilities.o \

```

```

common/fsm.o tcp/tcp-abs.o \
diffusion/diffusion.o diffusion/diff_rate.o diffusion/diff_prob.o \
diffusion/diff_sink.o diffusion/flooding.o diffusion/omni_mcast.o \
diffusion/hash_table.o diffusion/routing_table.o diffusion/iflist.o \
tcp/tfrc.o tcp/tfrc-sink.o mobile/energy-model.o apps/ping.o tcp/tcp-rfc793edu.o \
queue/rio.o queue/semantic-rio.o tcp/tcp-sack-rh.o tcp/scoreboard-rh.o \
plm/loss-monitor-plm.o plm/cbr-traffic-PP.o \
linkstate/hdr-ls.o \
mpls/classifier-addr-mpls.o mpls/ldp.o mpls/mpls-module.o \
routing/rtrmodule.o classifier/classifier-hier.o \
routing/addr-params.o \
routealgo/rnode.o \
routealgo/bfs.o \
routealgo/rbitmap.o \
routealgo/rlookup.o \
routealgo/routealgo.o \
diffserv/dsred.o diffserv/dsredq.o \
diffserv/dsEdge.o diffserv/dsCore.o \
diffserv/dsPolicy.o diffserv/ew.o \
queue/red-pd.o queue/pi.o queue/vq.o queue/rem.o \
queue/gk.o \
pushback/rate-limit.o pushback/rate-limit-strategy.o \
pushback/ident-tree.o pushback/agg-spec.o \
pushback/logging-data-struct.o \
pushback/rate-estimator.o \
pushback/pushback-queue.o pushback/pushback.o \
common/parentnode.o trace/basetrace.o \
common/simulator.o asim/asim.o \
common/scheduler-map.o common/splay-scheduler.o \
linkstate/ls.o linkstate/rtrProtoLS.o \
pgm/classifier-pgm.o pgm/pgm-agent.o pgm/pgm-sender.o \
pgm/pgm-receiver.o mcast/rcvbuf.o \
mcast/classifier-lms.o mcast/lms-agent.o mcast/lms-receiver.o \
mcast/lms-sender.o \
$(OBJ_STL)

# don't allow comments to follow continuation lines

# mac-csma.o mac-multihop.o \
# sensor-nets/landmark.o mac-simple-wireless.o \
# sensor-nets/tags.o sensor-nets/sensor-query.o \
# sensor-nets/flood-agent.o \

# what was here before is now in emulate/
OBJ_C =

OBJ_COMPAT = $(OBJ_GETOPT) common/win32.o
#XXX compat/win32x.o compat/tkConsole.o

```

```

OBJ_EMULATE_CC = \
emulate/net-ip.o \
emulate/net.o \
emulate/tap.o \
emulate/ether.o \
emulate/internet.o \
emulate/ping_responder.o \
emulate/arp.o \
emulate/icmp.o \
emulate/net-pcap.o \
emulate/nat.o \
emulate/iptables.o \
emulate/tcptap.o

OBJ_EMULATE_C = \
emulate/inet.o

OBJ_GEN = $(GEN_DIR)version.o $(GEN_DIR)ns_tcl.o $(GEN_DIR)ptypes.o

SRC = $(OBJ_C:.o=.c) $(OBJ_CC:.o=.cc) \
$(OBJ_EMULATE_C:.o=.c) $(OBJ_EMULATE_CC:.o=.cc) \
common/tclAppInit.cc common/tkAppInit.cc $(OBJ_CPP:.o=.cpp)

OBJ = $(OBJ_C) $(OBJ_CC) $(OBJ_GEN) $(OBJ_COMPAT) $(OBJ_PROTOLIB_CPP) $(OBJ_AGENTJ_CPP)

CLEANFILES = ns nse nsx ns.dyn $(OBJ) $(OBJ_EMULATE_CC) \
$(OBJ_EMULATE_C) common/tclAppInit.o \
$(GEN_DIR)* $(NS).core core core.$(NS) core.$(NSX) core.$(NSE) \
common/ptypes2tcl common/ptypes2tcl.o

SUBDIRS=\
indep-utils/cmu-scen-gen/setdest \
indep-utils/webtrace-conv/dec \
indep-utils/webtrace-conv/epa \
indep-utils/webtrace-conv/nlanr \
indep-utils/webtrace-conv/ucb

BUILD_NSE = nse

all: $(NS) $(BUILD_NSE) all-recursive

all-recursive:
for i in $(SUBDIRS); do ( cd $$i; $(MAKE) all; ) done

$(NS): $(OBJ) common/tclAppInit.o Makefile
$(LINK) $(LDFLAGS) $(LDOUT)$@ \
common/tclAppInit.o $(OBJ) $(LIB)

```

```

Makefile: Makefile.in
@echo "Makefile.in is newer than Makefile."
@echo "You need to re-run configure."
false

$(NSE): $(OBJ) common/tclAppInit.o $(OBJ_EMULATE_CC) $(OBJ_EMULATE_C)
$(LINK) $(LDFLAGS) $(LDOUT)$@ \
common/tclAppInit.o $(OBJ) \
$(OBJ_EMULATE_CC) $(OBJ_EMULATE_C) $(LIB)

ns.dyn: $(OBJ) common/tclAppInit.o
$(LINK) $(LDFLAGS) -o $@ \
common/tclAppInit.o $(OBJ) $(LIB)

PURIFY = purify -cache-dir=/tmp
ns-pure: $(OBJ) common/tclAppInit.o
$(PURIFY) $(LINK) $(LDFLAGS) -o $@ \
common/tclAppInit.o $(OBJ) $(LIB)

NS_TCL_LIB = \
tcl/lib/ns-compat.tcl \
tcl/lib/ns-default.tcl \
tcl/lib/ns-errmodel.tcl \
tcl/lib/ns-lib.tcl \
tcl/lib/ns-link.tcl \
tcl/lib/ns-mobilenode.tcl \
tcl/lib/ns-sat.tcl \
tcl/lib/ns-cmutrace.tcl \
tcl/lib/ns-node.tcl \
tcl/lib/ns-rtmodule.tcl \
tcl/lib/ns-hiernode.tcl \
tcl/lib/ns-packet.tcl \
tcl/lib/ns-queue.tcl \
tcl/lib/ns-source.tcl \
tcl/lib/ns-nam.tcl \
tcl/lib/ns-trace.tcl \
tcl/lib/ns-agent.tcl \
tcl/lib/ns-random.tcl \
tcl/lib/ns-namsupp.tcl \
tcl/lib/ns-address.tcl \
tcl/lib/ns-intserv.tcl \
tcl/lib/ns-autoconf.tcl \
tcl/rtp/session-rtp.tcl \
tcl/lib/ns-mip.tcl \
tcl/rtglib/dynamics.tcl \
tcl/rtglib/route-proto.tcl \
tcl/rtglib/algo-route-proto.tcl \
tcl/rtglib/ns-rtProtoLS.tcl \

```

```

        tcl/interface/ns-iface.tcl \
tcl/mcast/BST.tcl \
        tcl/mcast/ns-mcast.tcl \
        tcl/mcast/McastProto.tcl \
        tcl/mcast/DM.tcl \
tcl/mcast/srm.tcl \
tcl/mcast/srm-adaptive.tcl \
tcl/mcast/srm-ssm.tcl \
tcl/mcast/timer.tcl \
tcl/mcast/McastMonitor.tcl \
tcl/mcast/mftp_snd.tcl \
tcl/mcast/mftp_rcv.tcl \
tcl/mcast/mftp_rcv_stat.tcl \
tcl/mobility/dsdv.tcl \
tcl/mobility/dsr.tcl \
        tcl/ctr-mcast/CtrMcast.tcl \
        tcl/ctr-mcast/CtrMcastComp.tcl \
        tcl/ctr-mcast/CtrRPCComp.tcl \
tcl/rlm/rlm.tcl \
tcl/rlm/rlm-ns.tcl \
tcl/session/session.tcl \
tcl/lib/ns-route.tcl \
tcl/emulate/ns-emulate.tcl \
tcl/lan/vlan.tcl \
tcl/lan/abslan.tcl \
tcl/lan/ns-ll.tcl \
tcl/lan/ns-mac.tcl \
tcl/webcache/http-agent.tcl \
tcl/webcache/http-server.tcl \
tcl/webcache/http-cache.tcl \
tcl/webcache/http-mcache.tcl \
tcl/webcache/webtraf.tcl \
tcl/webcache/empweb.tcl \
tcl/webcache/empftp.tcl \
tcl/plm/plm.tcl \
tcl/plm/plm-ns.tcl \
tcl/plm/plm-topo.tcl \
tcl/mpls/ns-mpls-classifier.tcl \
tcl/mpls/ns-mpls-ldpagent.tcl \
tcl/mpls/ns-mpls-node.tcl \
tcl/mpls/ns-mpls-simulator.tcl \
tcl/lib/ns-pushback.tcl \
tcl/lib/ns-srcrt.tcl \
tcl/mcast/ns-lms.tcl \
tcl/lib/ns-qsnode.tcl \
$(NS_TCL_LIB_STL)

$(GEN_DIR)ns_tcl.cc: $(NS_TCL_LIB)
$(TCLSH) bin/tcl-expand.tcl tcl/lib/ns-lib.tcl $(NS_TCL_LIB_STL) | $(TCL2C) et_ns_lib > $@

```

```

$(GEN_DIR)version.c: VERSION
$(RM) $@
$(TCLSH) bin/string2c.tcl version_string < VERSION > $@

$(GEN_DIR)ptypes.cc: common/ptypes2tcl common/packet.h
./common/ptypes2tcl > $@

common/ptypes2tcl: common/ptypes2tcl.o
$(LINK) $(LDFLAGS) $(LDOUT)$@ common/ptypes2tcl.o

common/ptypes2tcl.o: common/ptypes2tcl.cc common/packet.h

libagentj.so: $(OBJ) common/tclAppInit.o
$(LINK) $(AGENTJ_SHARED_LDFLAGS) -o $@ common/tclAppInit.o $(OBJ) $(LIB)
mv libagentj.so $(AGENTJ_LIB_DIR)

install: force install-ns install-man install-recursive

install-ns: force
$(INSTALL) -m 555 -o bin -g bin ns $(DESTDIR)$(BINDEST)

install-man: force
$(INSTALL) -m 444 -o bin -g bin ns.1 $(DESTDIR)$(MANDEST)/man1

install-recursive: force
for i in $(SUBDIRS); do ( cd $$i; $(MAKE) install; ) done

clean:
$(RM) $(CLEANFILES)

AUTOCONF_GEN = tcl/lib/ns-autoconf.tcl
distclean: distclean-recursive
$(RM) $(CLEANFILES) Makefile config.cache config.log config.status \
    autoconf.h gnuc.h os-proto.h $(AUTOCONF_GEN); \
$(MV) .configure .configure- ;\
echo "Moved .configure to .configure-"

distclean-recursive:
for i in $(SUBDIRS); do ( cd $$i; $(MAKE) clean; $(RM) Makefile; ) done

tags: force
ctags -wtd *.cc *.h webcache/*.cc webcache/*.h dsdv/*.cc dsdv/*.h \
dsr/*.cc dsr/*.h webcache/*.cc webcache/*.h lib/*.cc lib/*.h \
../Tcl/*.cc ../Tcl/*.h

TAGS: force
etags *.cc *.h webcache/*.cc webcache/*.h dsdv/*.cc dsdv/*.h \
dsr/*.cc dsr/*.h webcache/*.cc webcache/*.h lib/*.cc lib/*.h \

```

```

../Tcl/*.cc ../Tcl/*.h

tcl/lib/TAGS: force
( \
cd tcl/lib; \
$(TCLSH) ../../bin/tcl-expand.tcl ns-lib.tcl | grep '^### tcl-expand.tcl: begin' | awk '{p
etags --lang=none -r '/^[ \t]*proc[ \t]+\([^ \t]+\)/\1/' 'cat .tcl_files'; \
etags --append --lang=none -r '/^\([A-Z][^ \t]+\)[ \t]+\(instproc|proc)\[ \t]+\([^ \t]+\)'
)

depend: $(SRC)
$(MKDEP) $(CFLAGS) $(INCLUDES) $(SRC)

srctar:
@cwd='pwd' ; dir='basename $$cwd' ; \
name=ns-'cat VERSION | tr A-Z a-z' ; \
tar=ns-src-'cat VERSION'.tar.gz ; \
list="" ; \
for i in 'cat FILES' ; do list="$${list} $$name/$$i" ; done; \
echo \
"(rm -f $$tar; cd .. ; ln -s $$dir $$name)" ; \
(rm -f $$tar; cd .. ; ln -s $$dir $$name) ; \
echo \
"(cd .. ; tar cfh $$tar [lots of files])" ; \
(cd .. ; tar cfh - $$list) | gzip -c > $$tar ; \
echo \
"rm ../$$name; chmod 444 $$tar" ; \
rm ../$$name; chmod 444 $$tar

force:

test: force
./validate

# Create makefile.vc for Win32 development by replacing:
# "# !include ..." -> "!include ..."
makefile.vc: Makefile.in
$(PERL) bin/gen-vcmake.pl < Makefile.in > makefile.vc
# $(PERL) -pe 's/^# (\!include)/\!include/o' < Makefile.in > makefile.vc

```

2.5.3 What's included in the *Agentj* Release?

The *Agentj* distribution consists of several co-operating software stacks, which are described in the following chapters of the manual. It includes the PAI interface to Protolib (described in Chapt. 4 and the Java NS2 agent extensions, described in Chapt. ??). *Agentj* also has an accompanying package, called P2PSX, which provides a P2P framework within NS2 [1].

2.6 Configuration

Agentj has fixed a number of configuration issues for this release. **Agentj** now uses the standard environment variables to configure the location of the agentj shared library (for JNI) and for the Java classpath required by the package. There are a couple of points, however:

- **JNI Configuration:** On my apple Mac, dynamic libraries for JNI are named specifically for this purpose, using the format `lib<libname>.jnilib`. This format is required, otherwise, the Java implementation cannot use the `LD_LIBRARY_PATH` environment variable to find the shard library (we use this in this release of **Agentj**). However, for ports to other platforms you will have to use the format required for that platform and include the necessary libraries required by the JNI interface. This is normally straight forward and information can be found in the Java Tutorial [14] for settings for the various platforms.
- **Classpath:** The `CLASSPATH` environment variable is used to initialise the JVM that is created by **Agentj**. Therefore, ANY paths required by your Java application running within NS2 will need to be specified using this environment variable. Since, we dynamically create a JVM from within the C++ code, this is really the only reliable mechanism for setting the `CLASSPATH` within the JVM. Normally, one would tend to specify these on the command line dynamically but this is not possible with **Agentj**. Further, if you build your own **Agentj** objects, then the classpaths for these will need to be included.

2.7 AgentjLogging

The `AGENTJDEBUG` environment variable is a course-grained mechanism which can be used to turn logging on or off. However, if turned on, `log4j` can be configured in a number of ways. **Agentj** uses a simple Java package, called `Autolog`, which is used to extend the `log4j` discovery mechanisms, allowing the user to specify local `log4j` configuration files, outside the scope of the classpath. This is described in more detail in the next two sections.

2.7.1 AutoLog Overview

`AutoLog` is a simple interface to initialise the `Log4j` logging system, which extends the discovery mechanisms for XML configuration files and provides

an auto configuration mode when there is no default configuration chosen by the user. In short, Autolog always tries to make the best of any particular environment.

The log4j library does not make any assumptions about its environment. In particular, there are no default log4j appenders, which results in an Error like the following:

```
log4j:WARN No appenders could be found for logger (MyApp).
log4j:WARN Please initialize the log4j system properly.
Process terminated with exit code 0
```

then, thereafter, no logging message are output. AutoLog tries to configure the login system under these circumstances to use the PatternLayout and to set the logging to the WARN level. However, it does more than this. The next section describes the discovery procedure.

2.7.2 AutoLog Discovery

The following procedure takes place:

- **Log4J:**

The default log4j initialization procedure is attempt. This searches the CLASSPATH for configuration files which you set using the Java property “log4j.configuration”. You can set this at the command-line using something like:

```
java -Dlog4j.configuration=myfile.xml MyClass
```

Note that the file you specify **MUST** be located somewhere within your CLASSPATH.

- **Autolog:** The autolog Java property is tried. This allows you to specify an absolute path/URL to your filename containing your XML configuration. This property is called “log4j.configuration.file”. You can set this using something like this:

```
java -Dlog4j.configuration.file=myfile.xml MyClass
```

- **Default:** If the other two fail, then we resort to apply a default appender to all loggers created within your code, which uses the Pattern layout using the following format:

```
%-7p: %1%nMESSAGE: %m (%d)%n%n
```

which results in your logging statement being output in the following format

```
WARN    : examples.SimpleLogging.<init>(SimpleLogging.java:19)
MESSAGE: Here is some WARN (2004-06-30 11:42:27,037)

FATAL   : examples.SimpleLogging.<init>(SimpleLogging.java:20)
MESSAGE: Here is some FATAL (2004-06-30 11:42:27,038)
```

2.7.3 Example XML Configuration

In the autolog config directory, there are a number of example scripts. Here, we show an example script that gives the same output as the default logging format described above, for comparison:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">

<log4j:configuration xmlns:log4j='http://jakarta.apache.org/log4j/'>
  <appender name="ToTheScreen" class="org.apache.log4j.ConsoleAppender">
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern"
        value="%-7p: %1%nMESSAGE: %m (%d)%n %n"/>
    </layout>
  </appender>

  <category name="A0123456789">
    <priority value="info" />
  </category>

<root>
  <priority value="debug" />
  <appender-ref ref="ToTheScreen" />
</root>

</log4j:configuration>
```

2.8 Conclusion

This chapter described the installation of the core packages needed for P2PS-NS2. The Protolib library needs to be installed first, followed by the P2PS-NS2 installation. Both installation require the editing of the NS Makefile.in file in order to add the correct dependencies into NS2. P2PS-NS requires the installation of both the static and shared libraries for the NS2 executable and the JNI bindings describes later in this manual.

Part II

Agentj Design and Implementation

In this part, we describe the design and implementation of **Agentj**. We describe the underlying toolkit, Protoli and PAI and then describe how **Agentj** extends this behaviour into NS2 to provide a generic interface for simulating Java applications.

Chapter 3

Protolib

Protean Protocol Prototyping Library (Protolib) is a low-level communication, event dispatching and timing package that can be used on top of a network or within the NS-2 network simulator environment. Protolib is not so much a library as it is a toolkit. The goal of the Protolib is to provide a set of simple, cross-platform C++ classes that allow development of network protocols and applications that can run on different platforms and in network simulation environments. Although Protolib is principally for research purposes, the code has been constructed to provide robust, and efficient performance and adaptability to real applications.

Currently Protolib supports most Unix platforms (including MacOS X) and WIN32 platforms. The most recent version also supports building Protolib-based code for the ns-2 simulation environment. The OPNET simulation tool has also been supported in the past and could be once again with a small amount of effort.

3.1 An overview of Protolib

A typical usage scenario of Protolib is given in the Fig 3.2 . Here, a timer is set up to trigger every 100 milliseconds. Upon a trigger event, a C++ callback is invoked that allows the application developer to integrate an event action. In this example, the application sends a UDP message using Protolib. This communication mechanism can be achieved using a standard UDP call across a network or between two NS-2 nodes. When the packet is received by the receiver, another event is generated indicating that data has been received. This, in turn, calls a routine that allows the application to collect the data from the UDP port and process it in some way. The application interface between the Protolib and the P2P middleware abstracts the reliance

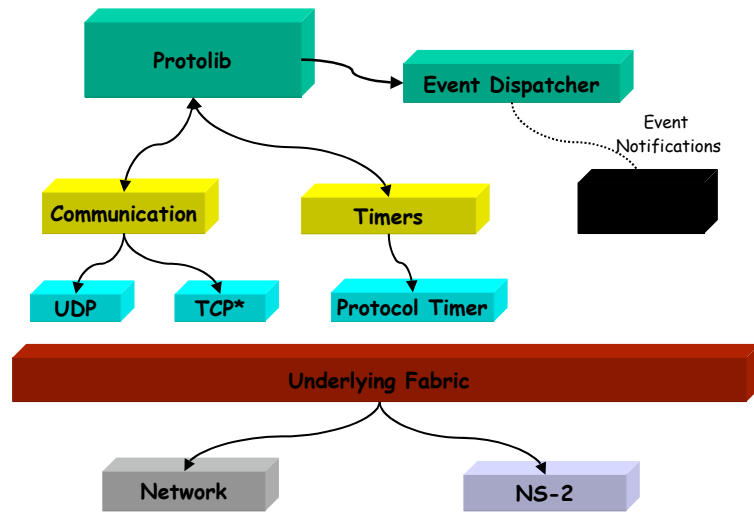


Figure 3.1: An overview of the Protolib toolkit, showing the three distinct components, sockets timers and a mechanism for dispatching events.

on specific networking/timing mechanisms in Protolib to create a generalized pluggable transport mechanism. Within PAI, middleware or indeed applications program to one interface and then choose the environment they want to run within e.g. Network or NS-2. This is very similar to GATLite but at a far lower level. PAI also support multiple sockets, timers and corresponding listeners for timeouts or UDP receive data events and establishes a cooperating event dispatching mechanism using multithreading.

3.2 Protolib Structure

The following classes are contained within the Protolib toolkit (taken from the descriptions provide in the release¹)

- **ProtoAddress:** Network address container class with support for IPv4, IPv6, and "SIM" address types. Also includes functions for name/address resolution.
- **ProtoSocket:** Network socket container class that provides consistent interface for use of operating system (or simulation environment) trans-

¹Given by Brian Adamson, email: adamson@itd.nrl.navy.mil

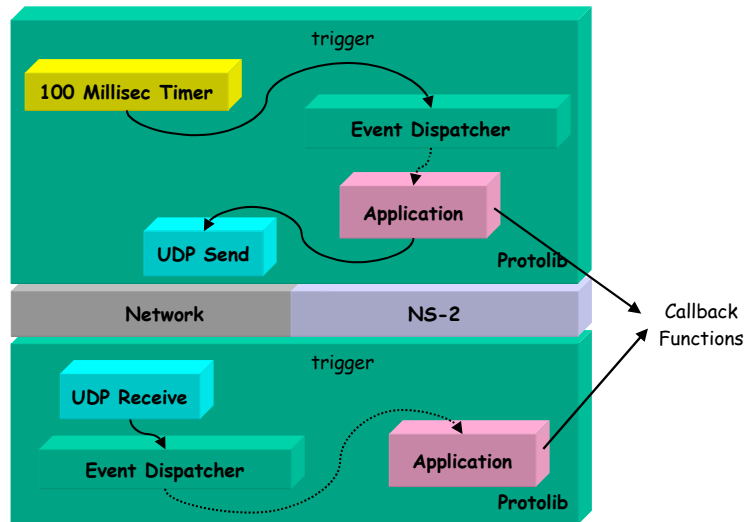


Figure 3.2: An overview of the functionality provided by the ProtoApp application, which triggers a data send once-per-second.

port sockets. Provides support for asynchronous notification to ProtoSocket::Listeners. The ProtoSocket class may be used stand-alone, or with other classes described below. A ProtoSocket may be instantiated as either a UDP or TCP socket.

- **ProtoTimer:** This is a generic timer class which will notify a ProtoTimer::Listener upon timeout.
- **ProtoTimerMgr:** This class manages ProtoTimer instances when they are "activated". The ProtoDispatcher (below) derives from this to manage ProtoTimers for an application. (The ProtoSimAgent base class contains a ProtoTimerMgr to similarly manage timers for a simulation instance).
- **ProtoTree:** Flexible implementation of a Patricia tree data structure. Includes a ProtoTree::Item which may be derived from or used as a container for whatever data structures and application may require.
- **ProtoRouteTable:** Class based on the ProtoTree Patricia tree to store routing table information. Uses the ProtoAddress class to store network routing addresses. It's a pretty dumbed-down routing table at

the moment, but may be enhanced in the future. Example use of the ProtoTree.

- **ProtoRouteMgr:** Base class for providing a consistent interface to manage operating system (or other) routing engines.
- **ProtoDispatcher:** This class provides a core around which Unix and Win32 applications using Protolib can be implemented. It's "Run()" method provides a "main loop" which uses the "select()" system call on Unix and the similar "MsgWaitForMultipleObjectsEx()" system call on Win32. It is planned to eventually provide some built-in support for threading in the future (e.g. the ProtoDispatcher::Run() method might execute in a thread, dispatching events to a parent thread).
- **ProtoApp:** Provides a base class for implementing Protolib-based command-line applications. Note that "ProtoApp" and "ProtoSimAgent" are designed such that subclasses can be derived from either to reuse the same code in either a real-world applications or as an "agent" (entity) within a network simulation environment (e.g. ns-2, OPNET). A "background" command is included for Win32 to launch the app without a terminal window.
- **ProtoSimAgent:** Base class for simulation agent derivations. Currently an ns-2 agent base class is derived from this, but it is possible that other simulation environments (e.g. OPNET, Qualnet) might be supported in a similar fashion.
- **NsProtoSimAgent:** Simulation agent base class for creating ns-2 instantiations of Protolib-based network protocols and applications.
- **ProtoExample:** Example class which derives either from ProtoApp or NsProtoSimAgent, depending upon compile-time macro definitions. It provides equivalent functionality in either the simulation environment or as a real-world command-line application. It demonstrates the use/operation of ProtoSocket based UDP transmission/reception, a ProtoTimer, and an example ProtoSocket-based TCP client-server exchange. (NOTE: TCP operation is not yet supported in the simulation environment. This will be completed in coming months. I plan to extend ns-2 TCP agents to support actual transfer of user data to support this.)
- **Other:** The Protolib code also includes some simple, general purpose debugging routines which can output to "stderr" or optionally log to a specified file. See "protoDebug.h" for details.

3.3 Conclusion

In this chapter, a brief overview of the Protolib toolkit was given. We gave an overview of its structure and the types of operations it is designed to support (e.g. UDP/TCP communication, timers and event dispatching) and illustrated this through the use of a simple but typical usage scenario. We then gave an overview of the key classes that make up the toolkit.

Protolib is an evolving toolkit. As PAI has been integrated on top of Protolib, it has been updated in order to support the functionality required by **Agentj** and its applications. PAI, described in the next chapter, focuses on providing the interfaces necessary for Java applications and middleware to use Protolib in a number of different contexts.

Chapter 4

The PAI Interface

The Protolib Application Interface (PAI) provides a layer that allows Protolib to be used Java applications. Specifically, it provides a generic interface to timers, sockets and dispatchers and then employs the use of the *factory method design pattern* in order to instantiate the required set of objects e.g. UDP sockets, TCP NS2 sockets, realtime timers, NS2 timers, NS2 event dispatchers etc.

The resulting PAI interface looks very similar to a Java interface. Where ever possibly, we have used the Java conventions for interfacing with the underlying objects. For example, instead of providing callback functions for a timer, we allow a user to attach multiple listeners to a timer in order to get notified when an event occurs. The resulting interface therefore is very simple and very Java friendly.

In this chapter, we give a brief overview of PAI and provide some programming examples of its use. We then show how PAI can be used within NS2 by providing a PAI NS2 agent for integrating with Protolib.

4.1 Overview of PAI

4.2 Programming PAI

4.3 Using PAI within NS2: The C++ Side

The Java PAI interface described in the last Chapter can be used directly by C++ applications within NS2. This chapters discusses the C++ agent classes that have been written for **Agentj** and how these can be used with PAI to pass data between NB2 nodes and to set off timers.

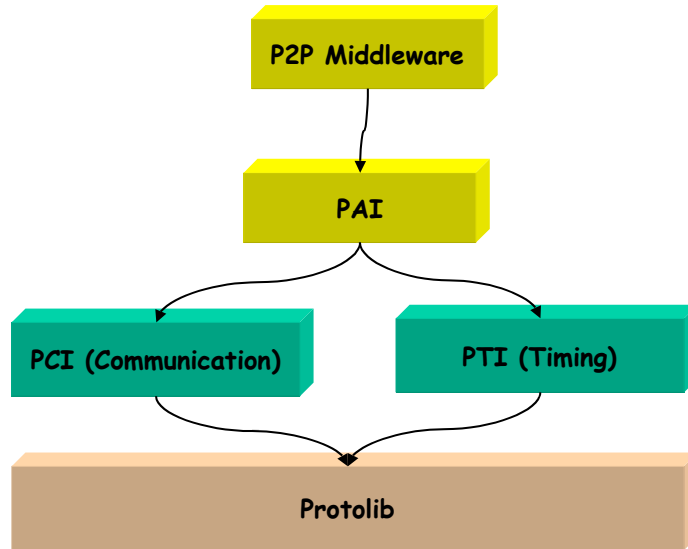


Figure 4.1: An overview of the PAI interface, showing the two sections to the underlying Protolib sockets and timers.

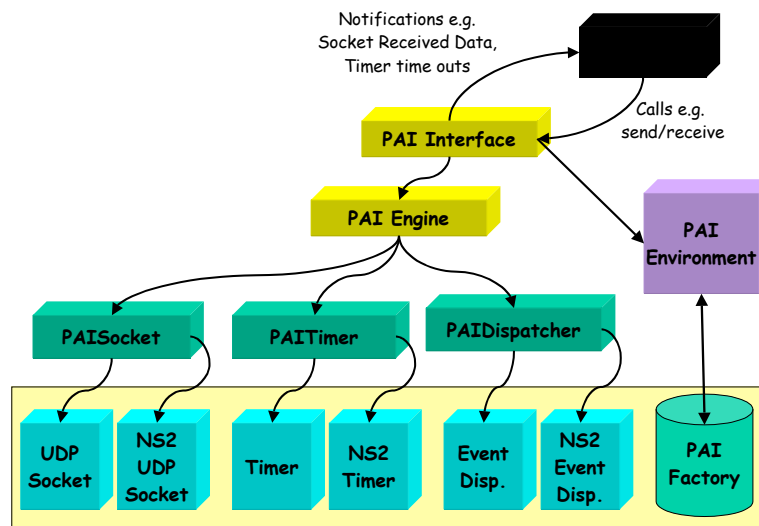


Figure 4.2: The PAI interface uses the Factory design pattern to create a common high level interface to whatever sockets or underlying timers the programmer is using.

When Timer times out:	When Data is Received:
<pre>void PAI_Example::OnTxTimeout() { pci->send(sock1, "127.0.0.1", buffer, len); }</pre>	<pre>void PAI_Example::OnSocketRecv() { char *buf = pci->recv(sock1, &addr, &len); }</pre>

Example Main Program:

```

pai.getEnvironment()->setBinding(PAI_NETWORK);
pai.getEnvironment()->setNetworkProtocol(PAI_UDP);

timer = pti->addTimer(1.0, 5);
sock = pci->addSocket(5004);

pci->addListener(sock, this, (CallbackFunc)&PAI_Example::OnTxTimeout);
pti->addListener(timer, this, (CallbackFunc)&PAI_Example::OnSocketRecv);

pti->runTimers();

```

Figure 4.3: An PAI code example, showing how you would implemented the standard Protolib demonstration, which sets a 1 second timer and sends data between two nodes.

4.4 Ns2 Agents

There are typically two main components within an NS2 scenario:

1. **C++ Agent:** This is used to represent the C++ behaviour within Ns2. Examples of agents could include transport protocols e.g. UDP agent, TCP agent etc. Agents can also be use to implement applications (using Protolib) or to broker them.
2. **TCL Script:** This sepcifies which NS2 C++ agents will be used and also to paint the scenario of the simulation you wish to perform within the simulation environment.

Figure 4.4 illustrates how these two components interacts within NS2. The TCL script sets up the environment, e.g., the communication links, the underlying network core, and specifies which agents are going to be used within the simulation. An example script is given below, which simply creates a BasicAgent class and invokes a 'hello' command on that agent.

```
set ns_ [new Simulator]
```

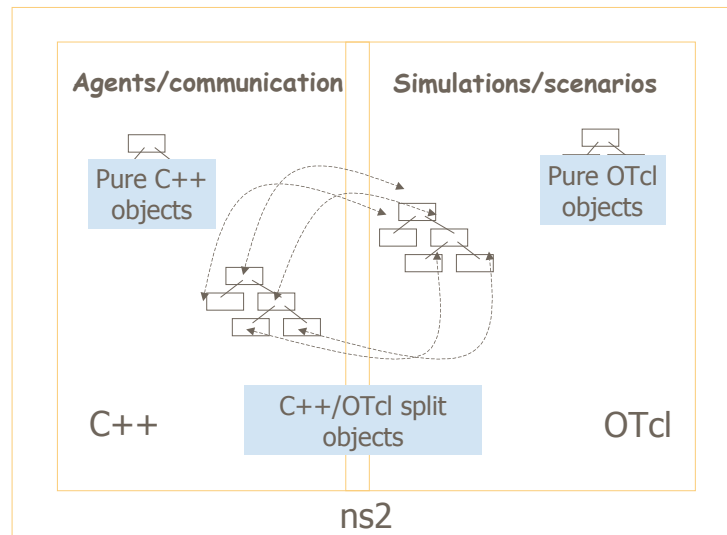


Figure 4.4: An overview of how TCL interacts with C++ agents within Ns2

```
# Create two nodes
set n1 [$ns_ node]
set n2 [$ns_ node]

# Put a link between them
$ns_ duplex-link $n1 $n2 64kb 100ms DropTail
$ns_ queue-limit $n1 $n2 100
$ns_ duplex-link-op $n1 $n2 queuePos 0.5
$ns_ duplex-link-op $n1 $n2 orient right

set p1 [new Agent/BasicAgent]
$ns_ attach-agent $n1 $p1

puts "Starting simulation ..."

$ns_ at 0.0 "$p2 hello"
$ns_ at 1.0 "finish $ns_"

proc finish {ns_} {
    $ns_ halt
    delete $ns_
}

$ns_ run
```

As you can see from the script, two Ns2 nodes are created and a network link is specified between them. We then create a custom Agent (called Basic Agent) and invoke a simple command on that agent. We are interested here in how such commands get passed to the agents. For specifics and a compendium of example scripts and agents can be found in the Ns-2 manual at the web site [12].

TCL scripts communicate with the C++ agents that they create by passing them commands. These commands get passes to a standardized method within the C++ Agent, called:

```
int PAIAgent::command(int argc, const char*const* argv) {
```

where, conventionally (as in `main()`), the arguments provide the following information:

- **argc:** specifies the number of strings representing the command and arguments which are being sent (i.e. `args 1` contains the command and 2 onwards specifies the arguments for the command).
- **argv:** These contain the actual strings representing the command and its arguments.

Therefore, a command, such as:

```
$ns_ at 0.0 "$p2 hello"
```

would be caught by the following command method implementation:

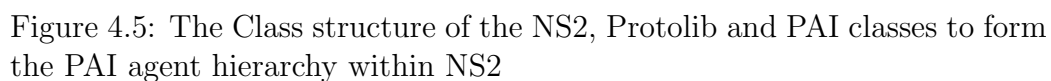
```
int PAIAgent::command(int argc, const char*const* argv) {

    if (2 == argc) {
        if (!strcmp(argv[1], "hello")) {
            cout << "BasicAgent: received a hello! " << endl;
            return TCL_OK;
        }
        // else if ...
    }

    // invoke the command from the next object up in the Agent class hierarchy

    return NsProtoAgent::command(argc, argv);
}
```

The PAI agents, described in the next section build upon this basic framework and Protolib to allow real applications to be run within the NS2 simulation environment.



4.5 PAI Agents and Protolib

4.6 Conclusion

Chapter 5

Agentj: Java Agents in NS2

In this chapter, we present the design and implementation of the **Agentj** framework and show the various levels at which **Agentj** interacts with other software packages and implements its functionality. In the previous two chapters, Protolib and the PAI interface were described which form the building blocks for **Agentj**. This chapter shows how these have been integrated through extensive use of the java Java Native Interface (JNI).

The first section provides an overview of the cooperating software components and the following section describes the core C++ and Java classes which have been used to implement these. Then, for completeness, we present the Java version of the PAI interface, which is at the core of this integration. Finally, we give a summary this integration by providing a complete overview of the software component interactions and the corresponding classes which implement the core behaviour of the system.

Although, the understanding of the details of this chapter is not absolutely necessary for **Agentj** users, it is highly recommended reading, as the details given here will give a broad understanding of the **Agentj** system and therefore, help a potential user in understanding what s/he can and cannot do.

5.1 Agentj Software Overview

Using **Agentj**, each C++ NS2 agent can (optionally) attach a Java agent. A Java NS2 agent is a Java program that can be accessed through the standard **Agentj** interface, which allows it to receive commands in the same ways as C++ NS2 agents do. Thus, specifically, an **Agentj** node is:

A Java object that implements the AgentJObject interface

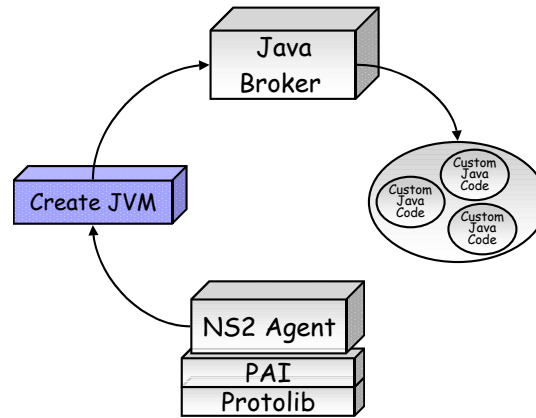


Figure 5.1: **Agentj** can attach any conforming Java object to an NS2 agent.

Agentj nodes can use any 3rd party Java application in order to implement the behaviour they require. Further **Agentj** nodes can use the **Agentj**'s PAI binding to access standard communication and timing classes in order to schedule events and discover and communicate with other **Agentj** nodes. In this fashion, complete Java simulations can be built up using these simple primitives.

For a Java application to become a **Agentj** node, it only needs to implement a simple Java interface and so the overhead of converting a Java application into the **Agentj** framework is minimal i.e. far easier than writing an Applet, for integration into another type of environment e.g. a Web browser. However, for such **Agentj** nodes to communicate with other nodes, they will need to interface with **Agentj**'s re-implementations of the normal Java classes they would use. This normally involves replacing all occurrences of *java.net.* with *pai.net.* We used precisely this procedure to integrate the P2PS middleware, discussed in [1].

In this section, we describe how the **Agentj** framework has been integrated using JNI and the core C++ Protolib and PAI toolkits and Java classes.

5.1.1 Creating **Agentj** Nodes

Figure 5.1 gives an overview of the software architecture employed by **Agentj** and shows how Java fits into this picture. Each NS2 *agent* can attach a Java object (i.e. Java agent) by accessing a Java Virtual machine (JVM) and by requesting that an association be made within the Java do-

main between the desired Java object and itself. This request results in a Java Hashtable being populated with an item pair; with the C++ agent pointer representing the key and **Agentj** Java object representing the object.

There could potentially be thousands of NS2 nodes and each one might want to instantiate and use a Java object. Therefore serious scalability issues can be encountered if this interaction is not slimline enough. In **Agentj**, the C++ JVM helper class (C2JBroker) therefore only allows **ONE** JVM to be created no matter how many nodes exist in the simulations. This JVM instantiates a singleton the *JavaBroker* class, which creates and manages the external Java objects. *JavaBroker* contains functionality that can dynamically create a Java object from a textual representation of its name (e.g. `pai.examples.NS2.SimpleCommand`).

Once created such objects are added to a local Hashtable, which associates an NS2 agent's *ID* with the associated object that has been created for this interaction. The NS2 agent's *ID* is actually its C++ pointer, which is reused later within the JNI binding (see below). Therefore, each NS2 node only instantiates the Java class it needs rather than any other wrapping classes. This implementation therefore maps one-to-one between the C++ NS2 agent and its corresponding Java object and therefore keeps the memory allocation to an absolute minimum; that is, we do not create thousands of *JavaBroker* objects, rather, we create one and have this act as a central locator for all Java objects.

The Java Hashtable is a static Class member (of *JavaBroker.java*, see section 5.2) and therefore only one Hashtable exists for all Java agents. When a node wishes to send a message to its attached Java object, it must first locate this object by searching this Hashtable using the pointer to the C++ agent. Once it has obtained a reference to the Java agent, it can forward the command to the object. This searching overhead is necessary for the issues discussed above in addressing scalability.

5.1.2 Inter-Agentj Communication

Once an **Agentj** node has been created and attached to its C++ counterpart, it can then use the supported **Agentj** communication and timing protocols in order to schedule events and communicate with other **Agentj** nodes. **Agentj** nodes do this by using **standard Java interfaces** which have been re-implemented in order to bind to the simulation world within NS2. These implementations include:

- **UDP Transport:** DatagramSocket and Multicast Socket have been rebound to PAI and Protolib to work within NS2.

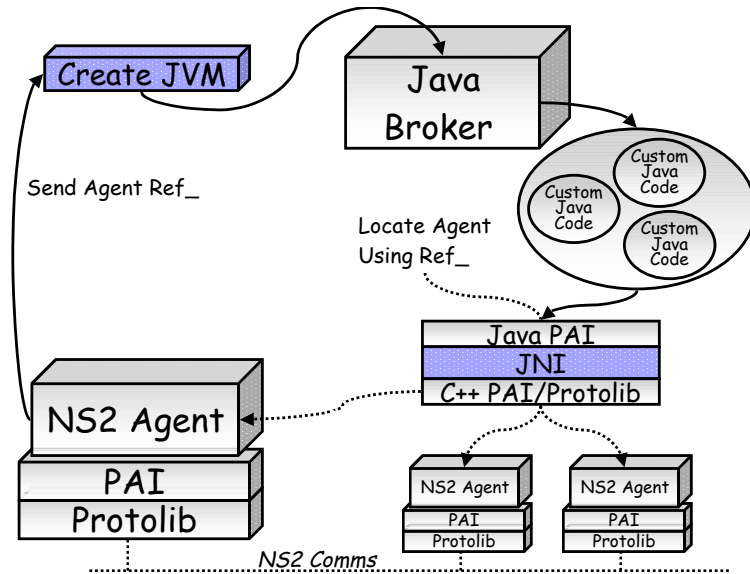


Figure 5.2: An overview of the *Agentj* software interaction between NS2, JNI, PAI and Protolib.

- **TCP Transport:** under construction....
- **Inet Support:** a number of the `InetAddress` methods have been implemented to work within the NS2 context.
- **Timing Libraries:** simple interfaces to timing functions have been implemented so that non realtime events (i.e. NS timing events) can be triggered at specific intervals during the simulation.

This functionality has been implemented by re-implementing the standard Java interfaces to the networked Java counterparts of these functions. Therefore, in order to use these within your Java program for example, you simply need to import the *pai.net* versions of `DatagramSocket` and `MulticastSocket` rather than the *java.io* versions.

Figure 5.2 gives an overview of the lower layers of this integration and provides a complete picture of how *Agentj* uses Java within its implementation. Here, the *Agentj* nodes interface through the Java PAI libraries by using a JNI binding to the C++ PAI and underlying Protolib toolkits. It is this software stack that describes the full integration. The standard Java interfaces described above merely provide convenient access to these libraries.

Therefore, an **Agentj** node interacts with other **Agentj** nodes by binding the standard Java interfaces to the PAI socket and timing classes within the JNI implementation, as indicated here. This integration allows **Agentj** nodes to issue commands to send data between NS2 nodes and to set up callbacks for timing events within NS2.

For this implementation, a JNI interface is provided between the Java PAI interface and the corresponding C++ PAI interface that contains the necessary underlying functionality. Within this implementation, the **Agentj** node's *ID* is used along within the JNI binding to the PAI interface in order to re-associate the Java object within its C++ Ns2 agent when we return back to the C++ context. In effect, what we are doing here is creating a JVM from C++, then we are using JNI to re-enter C++.

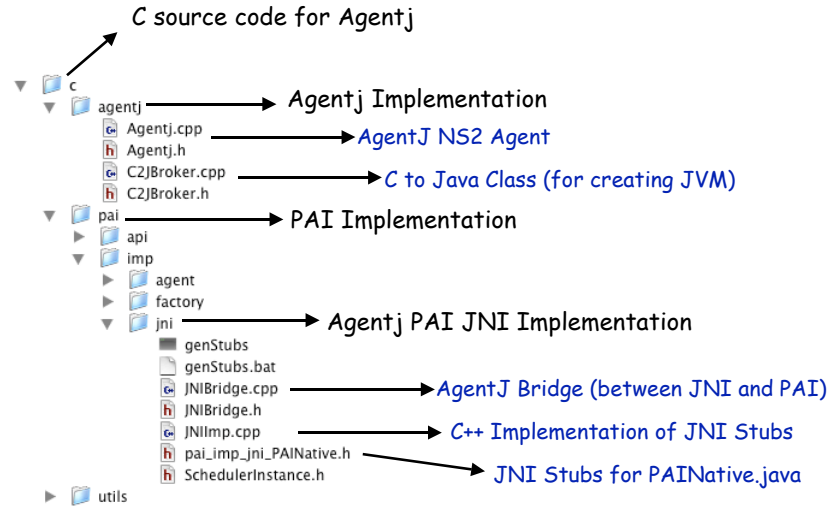
However, when we do this, JNI has no context and therefore we cannot use static methods to obtain pointers to parts of the NS2 system. Therefore, we have to transport these pointers manually through the JVM so we can re-establish our context when we are back in the C++ world. Specifically, the **Agentj** node needs to be able to reference its corresponding C++ NS2 agent so it can invoke the calls in the appropriate way. Otherwise, how would Protolib know which node to send the data from?

5.2 **Agentj** Implementation

This section gives an overview of the classes used to implement **Agentj** and describes the key classes that integrate the various components together. There are many underlying classes which are not described here but this section provides a good starting point for those interested in learning more about the integration.

Agentj is made up of a collection of C++ and Java classes. There are many more C++ classes than Java as the majority of the implementation is involved in binding between Protolib and higher level implementations and Java interfaces. For example, there are many housekeeping classes which provide lookup tables for mapping between C++ callbacks and Java Listener interfaces and vice versa.

Here, we provide descriptions for the key glue classes that tie the various parts of the system together between the C++ Java classes. The underlying C++ code for PAI and the specifics of these implementations are out of the scope of this chapter.

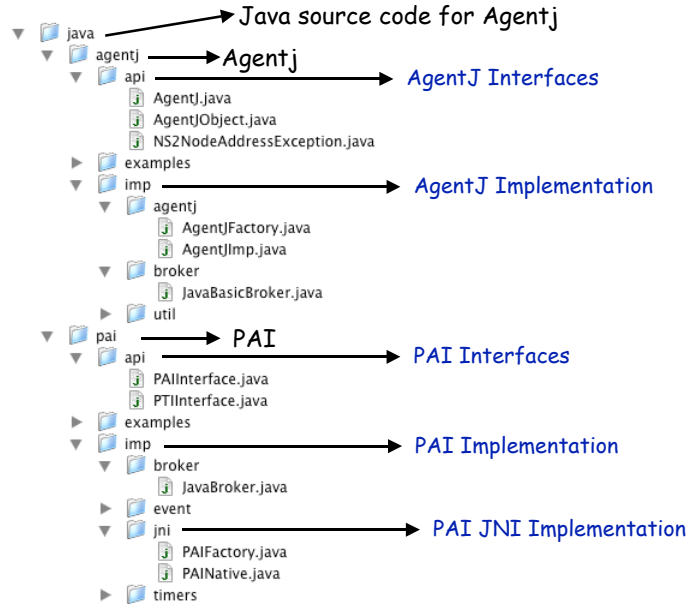
Figure 5.3: The C++ Classes within **Agentj**.

5.2.1 Organization of **Agentj** Classes

The **Agentj** directory tree is organized as if it was a Java application. Therefore, within this **Agentj** directory, there is a classes directory (where all classes live), a lib directory (for JAR files plus shared libraries), a doc directory (containing this manual) and a src directory (for source files), amongst others.

The src directory also is organized as if it was a Java application also (for the Java parts AND the C++ parts). I apologise in advance for this for those C++ developers who follow different standards but I write C++ programs as if they were Java and organize them as such! I believe however, that the overall structure is organized sensibly and due to the nature of this integration it is far easier to maintain a coherency between the Java and C++ classes.

At the top-level, there is a 'java' directory and a 'c' directory. These two directory structures are illustrated in Figures 5.3 for C++ and 5.4 for Java. At the next level, both the Java and C source trees are split into two sub-directories, one containing the classes specific to the implementation of **Agentj** and the other to the supporting implementations of PAI, as

Figure 5.4: The Java Classes within **Agentj**.

illustrated.

Beyond this, there are subdirectories that correspond to the particular section of the overall implementation that those classes are involved with. Therefore, these directories are organized as if they were Java packages (and indeed, in Java, they ARE Java packages). Therefore, APIs (or Java interfaces to APIs) are always put in a directory called 'api' and implementations of such interfaces are always put in directories called 'imp'. Subsequently, implementations of different interfaces or APIs are inserted into different sub directories.

These two Figures will be references in the next section, when we discuss the specifics about the individual classes which are used to implement parts of the overall system.

5.2.2 Key Agentj Classes

The central class in **Agentj**, which implements the bridge between the C++ NS2 nodes and Java is *Agentj.cpp* (found in *Agentj/src/c/agentj*, see 5.3). *Agentj.cpp* inherits from Protolib's *NsProtoAgent*, which is a C++ NS2 agent, which can use the Protolib data transport implementation e.g. UDP

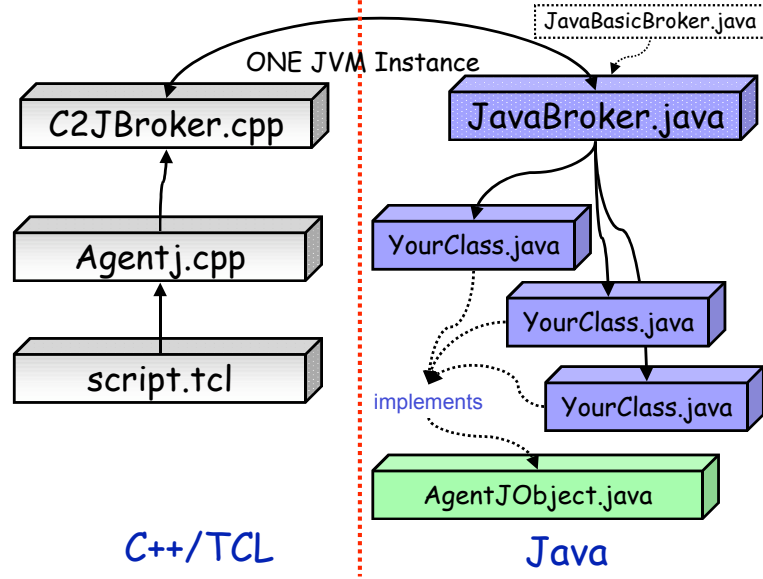


Figure 5.5: The C++ and Java classes used to implement the TCL/C++ and Java bridging mechanism.

and TCP, within NS2. *Agentj* uses the *Agentj* class to attach a Java Object to a C++ agent. Thereafter, the Java object itself interfaces through JNI to PAI, which in turn uses the Protolib to send the actual data packets.

As shown in Figure 5.5, *Agentj* uses the *C2JBroker* C++ class to create a Java Virtual Machine (JVM) and communicate with the singleton *JavaBroker* Java class. *C2JBroker* simply has these two functions: it creates a JVM and provides a C++ interface for sending messages to the *JavaBroker* Java class. However, it also implements some finer grained functionality. *C2JBroker* initializes *Agentj*'s environment variables. It parses the `LD_LIBRARY_PATH` and `CLASSPATH` variables and passes them to the JVM when it is being created so that the JVM can use the standard initialization procedures. It also, sets up the debugging settings i.e. by checking the `AGENTJDEBUG` and `AGENTJXMLCONFIG` variables (see Chapter 2).

The *JavaBroker* Java class allows provides a container for the Java objects that *Agentj* creates during the lifetime of the simulation. A Java Hashtable is used to store each Java object along with its identifier, which is the reference to the NS2 agent that this Java object belongs to. Each NS2 agent can attach (instantiate) one Java class and therefore there is a one-to-one interaction between an NS2 agent and its Java object.

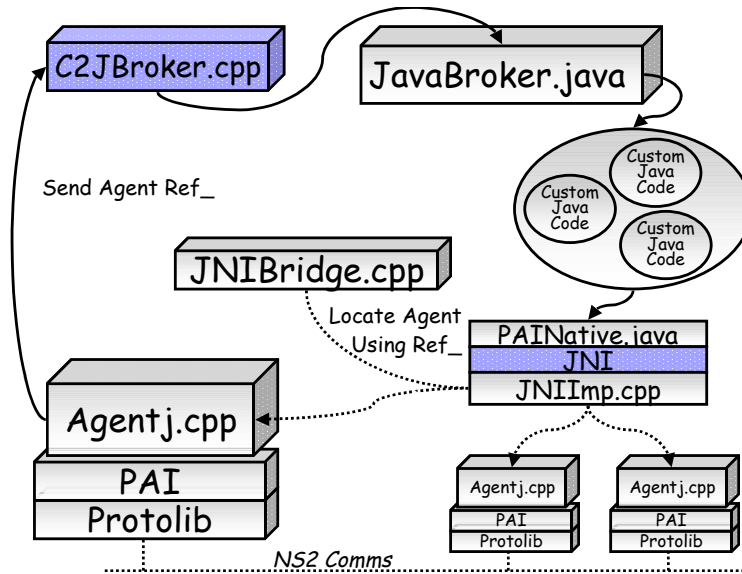


Figure 5.6: The C++ and Java classes shown in the broad overview of *Agentj*.

These interactions are shown in detail in Figure 5.5. Briefly, the programmer attaches the Java class within the TCL script. The *JavaAgent* then passes this data via *C2JBroker* to the *JavaBroker* Java object. The *JavaBroker* instantiates this object on-the-fly from the given class name. This means that you can attach multiple Java objects of the same type to every node or you are free to attach different objects to different NS2 nodes, depending on what you want to implement. For example, you could have a Java data collector agent talking to a Java data collection manager node instance. The only stipulation on the Java objects being created is that they should implement the *AgentJObject* interface.

Figure 5.6 shows the *Agentj* architecture given in Figure 5.2 but inserts the various C++ and Java classes which are used to implement each of the interactions. This clearly illustrates the interaction between the *Agentj*, *C2JBroker* and *JavaBroker* classes. Notice here that also the purpose of the *JavaBroker* is also illustrated; that is, it acts as a container for the multiple Java objects (*Agentj* nodes) that *Agentj* creates.

Also, depicted here, are the classes involved in the Java and JNI implementation of the PAI interface. The Java PAI interface is outline in the next subsection and is implemented by the *PAINative* Java class, which binds this interface, through JNI to the underlying C++ PAI implementation.

Within this implementation, there are two key C++ classes: *JNIImp* and *JNIBridge*. *JNIImp* is a direct implementation of the Java native methods contained within the *PAINative* Java class, whilst *JNIBridge* provides a persistent C++ object for storing information about the state of the C++ PAI interface during this session.

For example, *JNIBridge* contains the list of Java to C++ mappings of the various listeners that have been attached to the numerous sockets and timers which may have been created by the Java application. It also implements the callbacks to Java, which in turn, result in an event being passed to the Java classes which have requested to be notified about such events. This is a somewhat complicated procedure because C++ callbacks to Java have no context, so they must invoke a static Java method using an identifier notifying it which socket or timer this event belongs to. From this information, the Java method can work out which listeners are interested in this event.

5.2.3 The Java PAI interface

The Java PAI implementation consists of two Java interfaces:

- **PAIInterface:** interfaces to the communication part of the PAI interface (i.e. the sockets).
- **PTIInterface:** interfaces to the timing part of the PAI interface (i.e. the timers).

The *PAIInterface* allows the user to create multiple sockets and allows multiple socket listeners to be attached to each socket. This functionality is necessary for Java applications and is illustrated in the P2PS implementation, described in [1]. The *PTIInterface* does the same for timers. However, it is **important to note** that a user of *Agentj* does not have to use these classes directly. In fact, a user should not use these classes directly! They should use the PAI implementations of the UDP, TCP and timing interfaces for the default Java interfaces to these classes e.g. if you want to create a UDP socket then you should create a *pai.net.DatagramSocket*, which behaves exactly the same as a *java.net.DatagramSocket* except that it works within NS2. The interfaces here merely describe the lower level interfaces and mechanisms that are used to implement the overall structure.

The Java PAI interface is defined using a collection of Java interfaces and uses the *Factory Method Design* pattern [23] in order to create the appropriate underlying implementation. This means that other implementations (e.g. a native Java implementation) could be implemented at a later date. The

application developer however, would not notice this code change because s/he is working with a consistent interface. The JPAI interface can be found in package *pai.api* in the Java source tree and is listed below:

```
public interface PAIInterface extends PTIInterface {
    void init();

    void addPAISocketListener(DatagramSocket sock, PAISocketListener listener);

    void removePAISocketListener(DatagramSocket sock, PAISocketListener listener);

    void open(DatagramSocket sock, int port) throws SocketException;

    DatagramSocket addSocket(int port) throws SocketException;

    void removeSocket(DatagramSocket sock) throws SocketException;

    void setReuseAddress(DatagramSocket sock, boolean on) throws SocketException;

    void setSendBufferSize(DatagramSocket sock, int size) throws SocketException;

    void setReceiveBufferSize(DatagramSocket sock, int size) throws SocketException;

    void setSoTimeout(DatagramSocket sock, int timeout) throws SocketException;

    void send(DatagramSocket sock, DatagramPacket p) throws IOException;

    void receive(DatagramSocket sock, DatagramPacket p) throws IOException;

    void close(DatagramSocket sock);

    void joinGroup(MulticastSocket sock, InetAddress mcastaddr) throws IOException;

    void leaveGroup(MulticastSocket sock, InetAddress mcastaddr) throws IOException;

    void setMulticast(MulticastSocket sock, boolean val);

    public InetAddress getByName(String host) throws UnknownHostException;

    public InetAddress getLocalHost();

    public boolean cleanUp();
    public boolean runBlock();
    public boolean runNonBlock();

    public void setNS2Node(String nodeID);

    public void setNS2Scheduler(String schedulerID);
```

```
}

```

Most of the calls are self-explanatory. PAI uses the Java conventions for naming the classes e.g. `DatagramSocket` and `MulticastSocket`, both found in the `java.net` package (see [14]). The PAI Java implementation reimplements the methods from these classes in order to use the PAI interface. This enables the PAI interface to provide the functionality but it leaves the Java interface that developers are familiar with the same. Therefore, to create a Java UDP socket, you simply instantiate a `DatagramSocket`, which in turn invokes PAI to create a C++ PAI socket, which in turn creates a Protolib socket.

The *PTIInterface* is much simpler:

```
public interface PTIInterface {
    PAITimer addTimer(double delay, int repeat);

    void removeTimer(PAITimer timer);

    void addPAITimerListener(PAITimer timerID, PAITimerListener listener);

    void removePAITimerListener(PAITimer timerID, PAITimerListener listener);

    boolean runTimers();
}
```

Here, we simply provide a mechanism for creating a simple timer and allow a Java application to attach multiple listeners to it.

This design carries the whole weight of the **Agentj** implementation. To an application, the use of the conventional Java interface for creating UDP sockets means that they require very little source code modification in order to use this PAI JNI binding here. For example, in order to get P2PS working (see [1]) with this interface, a new resolver was created for UDP. This was a direct copy of the Java UDP resolver code with the occurrences of *java.net* replaced with *pa.net*, which enables this new resolver to look in the appropriate place for the PAI `DatagramSocket` class. Everything else followed through the various layers and P2PS required no further modification at the transport level. This re-implementation of these base Java classes can be found in the *pai.net* package in the source tree.

5.3 Conclusion

In this chapter, an overview of the **Agentj** integration was given, from a conceptual perspective and a source-code perspective. We illustrated the design

and architecture of **Agentj** then described in detail the interaction between the C++ and Java sections of the system. We then delved into the Java and C++ classes and outlined the key classes that implement this functionality and further, outlined the directory structure for **Agentj** for reference. We then inserted the names of these classes back into the architectural overview to give a clear picture of the entire system and the key components thereof. Finally, we gave a brief outline of the Java PAI interface to illustrate the kind of functionality it provides.

Part III

Using *Agentj*

In this part, we describe **Agentj** from a user's perspective. A typical user of **Agentj** would be a programmer of a Java application who would like to simulate their distributed application within NS2. In this part therefore, we describe how this can be achieved and which interfaces need to be implemented in order to build a bridge between NS2 and your distributed Java application. A comprehensive example of such an integration can be found in the accompanying manual for P2PSX [1], which integrates a comprehensive P2P middleware system into NS2 using **Agentj**.

Chapter 6

Using *Agentj*

Using **Agentj**, a Java NS2 agent can be attached to an NS2 node and can be used to integrate any Java application. This chapter gives an overview of the interaction between the TCL scripts, the C++ NS2 agents and Java objects, which can be accessed from each NS2 node. The various code snippets are taken from the **Agentj** source tree and pointers are referenced relative to the installation directory, when provided.

Figure 6.1 shows an overview of the interaction between the C++ agents, the *JavaBroker* and the Java PAI bridge that enables this to be interfaced with the C++ PAI library. As discussed briefly in the previous chapter, the C++ *JavaAgent* passes the pointer to the C++ agent to the *JavaBroker* when it requests to create and attach a Java agent to the NS2 node.

The *JavaBroker* class uses this pointer to store the created Java object in a hashtable for lookup but also pass this references across to the JNI interface, when a Java object requires the use of the PAI interface. This enables the JNI interface to locate the node that created the Java object and therefore whom is indirectly issuing the commands, which ensures that the data being sent through the sockets is sent from the correct node. The PAI interface sends these commands to the Protolib library, which in turn, uses the Protolib NS2 UDP implementation to send the data between the NS2 nodes.

6.1 Invoking Java Agents from NS2 Agents

Figure 6.2 shows interaction between an agent and its associated Java class. The programmer who wishes to use this Java functionality within their NS2 simulations only needs to be concerned within their NS2 TCL script and their Java class that implements the behaviour. The relationship between

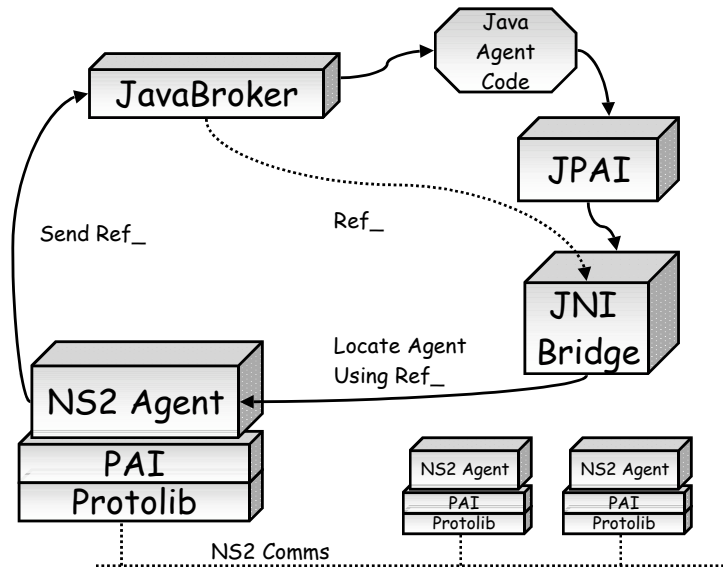


Figure 6.1: An overview PAI is accessed from within a Java node for an NS2 agent

an Ns2 agent and its Java class is very similar to the relationship between an NS2 TCL script and its associated C++ class (i.e. an NS2 agent) which implements the same kind of interaction through sending text commands between the two. The Java interface employs the same mechanism to bridge these different programming languages. The C++ agent (*JavaAgent*) simply acts as a go-between and passes that commands across to the appropriate Java object.

Therefore, the interface between the NS2 *JavaAgent* and the chosen Java Class it will interact with, uses the same command-style interface as the TCL-C++ interface for invoking functionality on NS2 agents. This command-style interaction satisfies some essential constraints:

- **Flexibility:** it will keep the flexibility of being able to use NS2 agents in any way programmer sees fit - the Java extensions are optional and any agent extending the *JavaAgent* can choose to use this functionality. However, the core C++ agent code can be programmed to incorporate and other functionality needed beyond the scope of Java.
- **Simplicity:** the scalability issues and framework for interacting with the Java objects can easily be hidden behind the container C++ and

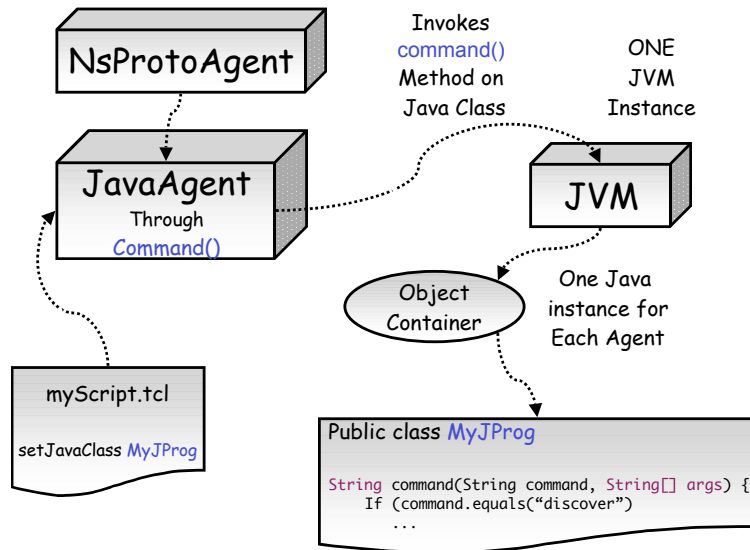


Figure 6.2: The interface to a Java program for an agent employs a similar interface to that of NS2 when communicating between the TCL scripts and the C++ classes.

Java classes - the programmer does not need to be aware of their presence.

- **Familiarity:** this mechanism allows communication between the NS2 agent and any attached Java class through the same familiar interface as NS2 programmers interface between the TCL scripts and C++ agents now.

This interaction is shown in Figure 6.3, which shows some *JavaAgent* commands for specifying and attaching a Java object and for sending it commands. These are the minimum commands needed in order to use your Java object. Each Ns2 node creates a Java object of its own choice by using the TCL command:

```
setClass <classpath> <class>
```

which allows the Java classpath to be set along with the name of the Java class to be instantiated for this NS2 node. Once the Java object has been created, commands can be sent by using the TCL command:

```
javaCommand <command> <args>
```

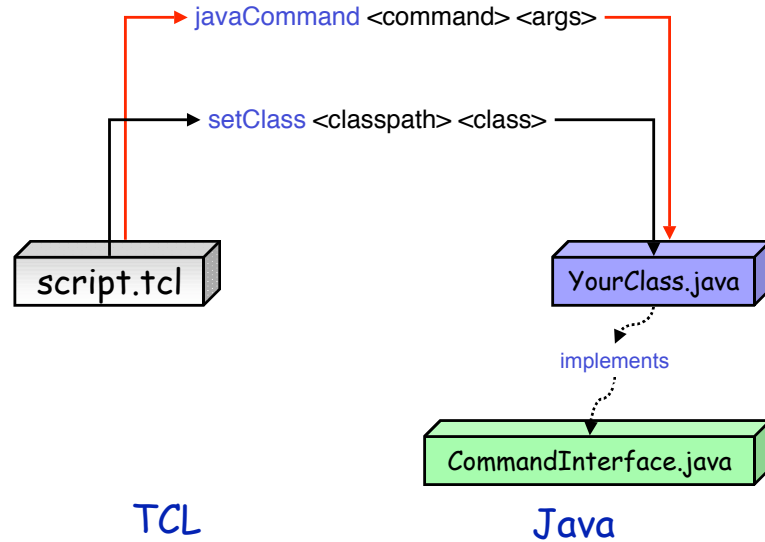


Figure 6.3: The user’s view of the interaction between the NS2 agent/script and the Java class associated with that NS2 node.

which would invoke the java command with the associated arguments. There are also other commands implemented that allow you to specify the delimiter to make it easier to chunk your arguments in a flexible way and for creating a trigger mechanism. The following 2 sections illustrate these commands through the use of example TCL and Java codes and the next chapter illustrates how you can extend the Java functionality to use PAI in order to send data between your Java objects through the NS2 subsystem.

6.2 Creating and Attaching a Java Agent

This is a *Hello World* example that demonstrates how to specify the Java classpath and choose a Java class to instantiate and attach to your C++ agent. It then implements a simple *hello* function which is invoke on the Java object.

6.2.1 The TCL Side

The following is the TCL script part of the implementation, which creates two *JavaAgent* NS2 nodes that each create a *SimpleCommand* Java object

and then invoke a 'hello' command on that object. This example can be found in `examples/pai/javaAgent/startJava.tcl`.)

```
puts "Starting..."

# Create a simulator instance
set ns_ [new Simulator ]

# Create two nodes
set n1 [$ns_ node]
set n2 [$ns_ node]

# Put a link between them
$ns_ duplex-link $n1 $n2 64kb 100ms DropTail
$ns_ queue-limit $n1 $n2 100
$ns_ duplex-link-op $n1 $n2 queuePos 0.5
$ns_ duplex-link-op $n1 $n2 orient right

puts "Creating JavaAgent NS2 agents and attach them to the nodes..."
set p1 [new Agent/JavaAgent]
$ns_ attach-agent $n1 $p1

set p2 [new Agent/JavaAgent]
$ns_ attach-agent $n2 $p2

puts "CREATED OK          .... .."

# Initialize each broker telling it what its NS2 address is

puts "In script: Initializing ..."

$ns_ at 0.0 "$p1 initAgent"
$ns_ at 0.0 "$p2 initAgent"

puts "Setting Java Object to use by each agent ..."

$ns_ at 0.0 "$p1 setClass
/Users/scmijt/Apps/nrl/p2ps-ns2/classes pai.examples.ns2.SimpleCommand"

$ns_ at 0.0 "$p2 setClass
/Users/scmijt/Apps/nrl/p2ps-ns2/classes pai.examples.ns2.SimpleCommand"

# send a message to each agent and tell it to print it to the screen
# This is a "HelloWorld" program for JavaAgents

$ns_ at 0.0 "$p1 javaCommand hello AStringFromP1"
$ns_ at 0.0 "$p2 javaCommand hello AStringFromP2"
```

```

$ns_ at 10.0 "finish $ns_"

proc finish {ns_} {
    $ns_ halt
    delete $ns_
}

$ns_ run

```

The Java agent parts can be seen in this example. The *setClass* function sets the classpath to the *p2ps-ns2* installations classes directory. Here, the actual java class that this node will be using is specified as *pai.examples.ns2.SimpleCommand*. Note here that you can load in classes that are contained in any java package that you wish as long as you follow the Java conventions for locating the compiled versions of these classes.

Once the Java classes have been located, you can then execute various commands by using the *javaCommand* instruction. Here we ask the Java class to execute a *hello* command and pass a string as an argument, identifying the node that is sending the message i.e. *AStringFromP1*. This simple example demonstrates that two Java objects have been created, one for each node and each Java object has been correctly associated or bound to the particular NS2 node.

6.2.2 The Java Side

On the java side of things each object you want to talk to must implement a standard interface called "CommandInterface" which enforces that every Java object implementing this interface implements this command method:

```

package pai.broker;

public interface CommandInterface {

    public String command(String command, String value);
}

```

Every class that you wish to be used from an NS2 agent must implement this Java interface so that it can understand the instructions that are sent to it. Below, an example Java class is given to illustrate the code involved in this process (the actual Java code for this and all other examples

can be found in the *src/jpai/pai/examples/ns2* directory):

```

package pai.examples.ns2;

```

```
import pai.broker.CommandInterface;

public class SimpleCommand implements CommandInterface {

    static int count=0;

    int myID;

    public SimpleCommand() {
        ++count;
        myID=count;
    }

    public String command(String command, String args[]) {

        if (command.equals("hello"))
            System.out.println("SimpleCommand(" + myID + ")
                called with Val: " + args[0]);

        return "All called ok from node " + myID;
    }
}
```

As you can see, this is extremely simple, the C++ and Java JVM class take care of all the complexity. In the command method, you can implement any behaviour you want. You can also return a *String* to your C++ program as indicated. This could allow you, for example, to discover other NS2 nodes using P2PS and then return their address to your C++ agent and keep the control at this point (helpful for non-java programmers!).

6.3 Changing the Command Delimiter

This example demonstrates how you would change the delimiter used to separate command arguments sent to your Java application. The default is to use a white space (as in NS2) to automatically parse the arguments and send them as a sequence of arguments to your agent or Java object. Within the Java NS2 implementation however, this choice is left up to the programmer. Therefore, you could specify for example a '-' symbol as a delimiter and a sequence such as this

8 - cherry apple oranges - to eat

would be parsed and sent to your program as 3 strings:

```

8
cherry apple oranges
to eat

```

This allows more flexibility in the way you send instructions to your Java code because it does not limit the input to contiguous strings. The example given below demonstrates how this is achieved from the TCL and Java sides.

6.3.1 The TCL Side

The following is the TCL script part of the implementation, which creates two *JavaAgent* NS2 nodes that each create a *ChangeDelimiter* Java object and then change the delimiter of one of the nodes in order to split up the input with respect to a '-' symbol. Note that setting delimiters is a global process and therefore can be set through any node and will be applied to all nodes. This example can be found in `examples/pai/javaAgent/changeDelimiter.tcl`

```

puts "Starting..."

# Create simulator instance
set ns_ [new Simulator]

# Create two nodes
set n1 [$ns_ node]
set n2 [$ns_ node]

# Put a link between them
$ns_ duplex-link $n1 $n2 64kb 100ms DropTail
$ns_ queue-limit $n1 $n2 100
$ns_ duplex-link-op $n1 $n2 queuePos 0.5
$ns_ duplex-link-op $n1 $n2 orient right

puts "Creating JavaAgent NS2 agents and attach them to the nodes..."
set p1 [new Agent/JavaAgent]
$ns_ attach-agent $n1 $p1

set p2 [new Agent/JavaAgent]
$ns_ attach-agent $n2 $p2

puts "In script: Initializing  ..."

$ns_ at 0.0 "$p1 initAgent"
$ns_ at 0.0 "$p2 initAgent"

puts "Setting Java Object to use by each agent ..."

```

```

$ns_ at 0.0 "$p1 setClass
/Users/scmijt/Apps/nrl/p2ps-ns2/classes pai.examples.ns2.ChangeDelimiter"

$ns_ at 0.0 "$p2 setClass
/Users/scmijt/Apps/nrl/p2ps-ns2/classes pai.examples.ns2.ChangeDelimiter"

# Delimiters are global and can be set through any node

$ns_ at 0.0 "$p1 javaCommand setDelimiter -"

$ns_ at 0.0 "$p2 javaCommand hello A-String-From-P2"

$ns_ at 10.0 "finish $ns_"

proc finish {ns_} {
    $ns_ halt
    delete $ns_
}

$ns_ run

```

The Java classes are located and instantiate as previous. Now, we can use the *javaCommand setDelimiter* instruction to change the delimiter. We set this to '-' and then send a single contiguous string to node p2 (A-String-From-P2) by using the 'hello' command. Now instead of passing this as a single string (as you would get in the NS2 C++ binding), you would get 4 separate string send to your program, which can be accessed individually, for example, as:

```

A
String
From
P2

```

6.3.2 The Java Side

On the java side *ChangeDelimiter.java* implements the "CommandInterface" to identify that it can process commands:

```

package pai.examples.ns2;

import pai.broker.CommandInterface;

public class ChangeDelimiter implements CommandInterface {

    static int count=0;

```

```

int myID;

public ChangeDelimiter() {
    ++count;
    myID=count;
}

public String command(String command, String args[]) {

    if (command.equals("hello")) {
        System.out.println("Command has "
            + args.length + " arguments");
        for (int i=0; i<args.length; ++i) {
            System.out.println("Arg[" + i + "] = " + args[i]);
        }

        return "All called ok from node " + myID;
    }
}

```

Here, the 'hello' command simply processes through the arguments and prints each to the screen on a separate line. Therefore, running the script will produce the following output:

```

In script: Initializing ...
Setting Java Object to use by each agent ...
Classpath is -Djava.class.path=/Users/scmijt/Apps/nrl/p2ps-ns2/classes
command has 4 arguments
Arg[0] = A
Arg[1] = String
Arg[2] = From
Arg[3] = P2

```

6.4 Conclusion

Then, two different examples were provided that illustrate how one would attach a Java object to an NS2 node and how one can execute Java commands on that object. Lastly, an example was given that demonstrates how you can change the delimiter used to parse the list of arguments you can send to your Java object. This employs a flexible mechanism that can use any string as

a delimiter to send lists or sentences to your Java object without having to parse further.

Chapter 7

Advanced Agentj

7.1 Agentj and PAI

In the last section, we discussed the way Java objects could be attached to Java agents and invoke from within NS2 simulations. In this chapter, an overview of how such Java nodes can be used to send packages between NS2 nodes by using the PAI interface, described in Chapt. 4. The Java interface contains a an interface to PAI through JNI that enables the Java objects to create sockets, attach listeners to the sockets and trigger events.

7.1.1 Using the Java PAI Interface in Ns2 Java Objects

Each Java objects that has been attached to an NS2 node must implement the *PAIAccessInterface* given below, which can be found in the `pai.broker` package within the source tree:

```
package pai.broker;

import pai.api.PAIInterface;

public interface PAIAccessInterface {

    public void setPAI(PAIInterface pai);
}
```

PAIAccessInterface provides a mechanism for the *JavaBroker* object to create a *PAIInterface* object to the JNI PAI implementation and pass this reference to your Java code. You can then use this reference directly to make PAI calls just as you would if you were using PAI directly.

This mechanism manages the creation and deletion of the PAI JNI implementation and sets variables in the JNI before each invocation so that it has the correct reference to the object it is dealing with at that moment. Briefly, the *JavaBaker* only create **one** instance of the PAI JNI implementation. This means that before each call it must set the reference to the actual NS2 node it is about to issue a command to enabling the interface to create the appropriate binding to PAI at the lower levels. This design adds a small overhead to each call but saves a substantial amount of memory since it efficiently uses one instance of the code rather than one for each node, which would increase memory consumption greatly (i.e. imagine if you had thousands of nodes).

7.2 Example 1: Sending Data From One Node to Another

This example uses the Java *PAICommands* class to send data between two NS2 nodes. The actual Java code specifies which nodes to communicate with. This simple example demonstrates how Java objects can be attached to an NS2 nodes and used to create sockets and send data between nodes.

7.2.1 The TCL Side

The following is the TCL script part of the implementation, which creates two *JavaAgent* NS2 nodes attaches the *PAICommands* Java object to them, initializes them and then sends data from the first node to the second by setting the NS 2 address of the second node directly from the script, using the *setSendTo* command.

```
# Create multicast enabled simulator instance
set ns_ [new Simulator]

# Create two nodes
set n1 [$ns_ node]
set n2 [$ns_ node]

# Put a link between them
$ns_ duplex-link $n1 $n2 64kb 100ms DropTail
$ns_ queue-limit $n1 $n2 100
$ns_ duplex-link-op $n1 $n2 queuePos 0.5
$ns_ duplex-link-op $n1 $n2 orient right

puts "Creating JavaAgent NS2 agents and attach them to the nodes..."
```

7.2. EXAMPLE 1: SENDING DATA FROM ONE NODE TO ANOTHER⁹⁹

```
set p1 [new Agent/JavaAgent]
$ns_ attach-agent $n1 $p1

set p2 [new Agent/JavaAgent]
$ns_ attach-agent $n2 $p2

puts "In script: Initializing agents  ..."

$ns_ at 0.0 "$p1 initAgent"
$ns_ at 0.0 "$p2 initAgent"

puts "Setting Java Object to use by each agent ..."

$ns_ at 0.0 "$p1 setClass
/Users/scmijt/Apps/nrl/p2ps-ns2/classes pai.examples.ns2.PAICommands"
$ns_ at 0.0 "$p2 setClass
/Users/scmijt/Apps/nrl/p2ps-ns2/classes pai.examples.ns2.PAICommands"

puts "Starting simulation ..."

$ns_ at 0.0 "$p1 javaCommand init"
$ns_ at 0.0 "$p2 javaCommand init"

$ns_ at 0.0 "$p1 javaCommand setSendTo [$n2 node-addr]"
$ns_ at 0.0 "$p1 javaCommand start"

$ns_ at 10.0 "finish $ns_"

proc finish {ns_} {
$ns_ halt
delete $ns_
}

$ns_ run
```

The Java classes are located and instantiate as described in Sect. 6.2. Now, we can use the *javaCommand init* instruction to initialize the Java nodes, set the send to address (the node where the data will be sent to) and start the node off, which results in it sending the data.

7.2.2 The Java Side

On the java side *PAICommands.java* implements various instructions to help send data and to trigger timers etc:

```
package pai.examples.ns2;
```

```
import pai.broker.CommandInterface;
import pai.broker.PAIAccessInterface;
import pai.api.PAIInterface;
import pai.net.DatagramSocket;
import pai.net.DatagramPacket;
import pai.net.InetAddress;
import pai.impl.PAITimer;
import pai.impl.Logging;
import pai.event.PAISocketEvent;
import pai.event.PAISocketListener;
import java.net.SocketException;
import java.io.IOException;

public class PAICommands implements CommandInterface, PAIAccessInterface,
                                   PAISocketListener {

    PAIInterface pai;
    String sendTo;
    DatagramSocket s;
    PAITimer t;
    int count=0;

    public void init() {
        try {
            s = pai.addSocket(5555);
            pai.addPAISocketListener(s,this);
        } catch (SocketException e) {
            System.out.println("Error opening socket");
        }
        catch (IOException ep) {
            System.out.println("Error opening socket");
        }
    }

    void start() {
        timerTriggered(); // transmit first packet right away
    }

    public void dataReceived(PAISocketEvent sv) {
        try {
            ++count;
            byte b[] = new byte[15];
            DatagramPacket p = new DatagramPacket(b,b.length);
            pai.receive(s, p);
            if (Logging.isEnabled()) {
                System.out.println("PAICommands: Received " +
                    " PACKET NUMBER -----> " + count);
                System.out.println("PAICommands: Received " +
                    + new String(p.getData()) +
```

7.2. EXAMPLE 1: SENDING DATA FROM ONE NODE TO ANOTHER101

```
        " from " + p.getAddress().getHostAddress());
    }
} catch (IOException ep) {
    System.out.println("PAICommands: Error opening socket");
}
}

public void timerTriggered() {
    try {
        byte b[] = (new String("Hello Proteus " +
            String.valueOf(count)).getBytes());
        DatagramPacket p =new DatagramPacket(b, b.length,
            new InetAddress(sendTo), 5555);
        pai.send(s,p);
    } catch (IOException eh) {
        System.out.println("Error Sending Data");
    }
}

public String command(String command, String args[]) {
    if (command.equals("init")) {
        init();
        return "OK";
    }
    else if (command.equals("setSendTo")) {
        sendTo = args[0];
        return "OK";
    }
    else if (command.equals("start")) {
        start();
        return "OK";
    }
    else if (command.equals("trigger")) {
        timerTriggered();
        return "OK";
    }
    else if (command.equals("cleanUp")) {
        pai.cleanUp();
        return "OK";
    }

    return "ERROR";
}

public void setPAI(PAIInterface pai) {
    this.pai=pai;
}
}
```

Firstly, you'll notice that *PAICommands* implements three interfaces:

- **CommandInterface:** so that it understands how to execute commands, as described in the previous chapter.
- **PAIAccessInterface:** (see `pai.broker.PAIAccessInterface`) this interface is a tagging mechanism that tells the subsystem that your Java object wishes to use the JNI interface. Without this, your object cannot use the efficient memory allocation that the subsystem provides for managing all Java objects. You could in principle access PAI directly but you'd have to manage pointers yourselves, which would be tedious. Using this interface, the *JavaBroker* notifies you of the instance of the *pai* interface by calling the implemented method from this interface, called **setPAI(PAIInterface pai)**, as illustrated. This allows you to store the *pai* reference locally and use it within your Java object.
- **PAISocketListener:** this allows your class to be notified when data arrives at a *PAISocket*. Briefly, within Java, you attach yourself (or attach others) as a listener on an object and this results in the notification of certain events when they arrive. To make the semantics clear, you have to implement an interface which enables the source object to notify you when its state changes. This is achieved generally by a listener interface, which *PAISocketListener* implements. Java listeners are an implementation of a callback mechanism. Within C++ you have to point to actual functions, which Java you attach listeners. The interface looks like this:

```
package pai.event;

public interface PAISocketListener {
    public void dataReceived(PAISocketEvent event);
}
```

which contains one method, *dataReceived* that gets invoked when data arrives at the socket. The *dataReceived* method passes a *PAISocketEvent*, which contains details about the socket that issued the event (i.e. you may be a listener to several sockets). Once this event is received, you can use *pai* to retrieve the data, using the *receive* method - which takes the socket as a parameter and a *DatagramPacket*, which is a container to hold the incoming data (this is the standard Java mechanism for doing this).

Briefly, the object is initialized by creating a socket on port 5555. We then add ourselves as a listener for events from this socket. The *start* method gets invoked when a *start* command is received from the NS2 TCL script, this simply invokes the trigger function, which results in a data packets being sent to the *sendTo* NS2 node. The *sendTo* variable is set using the *setSendTo* TCL command as described previously.

Within the *dataReceived* method, messages are printing out if logging is enabled. There is a static class in *pai.impl.Logging*, which is set globally for all classes within the JVM to turn on or off comments. If it is enabled then you get a verbose output - the default is that it is set to *on*.

7.3 Example 2: Using the Trigger Mechanism

This is a Java example, which implements the *ProtoApp* scenario, the demonstration class for Protolib. Briefly, a trigger is set off once a second to tell the Java object to send data to another node. When the data is received by the receiving NS-2 node, another Java method is triggered allowing it to read the data using the *PAISocketListener* interface.

The actual trigger mechanism is implemented in C++ but this then triggers a method in the Java object to tell it to read the data. This example also uses the *PAICommands* class. When the C++ trigger times out, it sends a 'trigger' command to the Java object, which results in the *timerTriggered()* method being called. This is equivalent functionality to *ProtoApp*, but in Java. However, the actual interface to the timer is set within the NS2 TCL script and not the C++ class, enabling the programmer to change the timer's parameters without having to recompile the whole of NS2.

7.3.1 The TCL Side

The following is the TCL script part of the implementation, which creates two *JavaAgent* NS2 nodes attaches the *PAICommands* Java object to them, initializes them and then sets up the node that will receive the data by invoking the *setSendTo* command on the first node - node 0 sends data to node 1 in this example. We then start a timer by using

```
$ns_ at 0.0 "$p1 startTimer 1 -1"
```

which sets off a timer that times out once per second and runs forever (i.e. -1 flag). The timer is stopped at the end of the simulation. Here is the TCL script:

```

# Create simulator instance
set ns_ [new Simulator]

# Create two nodes
set n1 [$ns_ node]
set n2 [$ns_ node]

# Put a link between them
$ns_ duplex-link $n1 $n2 64kb 100ms DropTail
$ns_ queue-limit $n1 $n2 100
$ns_ duplex-link-op $n1 $n2 queuePos 0.5
$ns_ duplex-link-op $n1 $n2 orient right

puts "Creating PAI Broker Agents ..."
# Create two Protean example agents and attach to nodes
set p1 [new Agent/JavaAgent]
$ns_ attach-agent $n1 $p1

set p2 [new Agent/JavaAgent]
$ns_ attach-agent $n2 $p2

puts "CREATED OK          ...."

# Initialize each broker telling it what its NS2 address is

puts "In script: Initializing ..."

$ns_ at 0.0 "$p1 initAgent"
$ns_ at 0.0 "$p2 initAgent"

$ns_ at 0.0 "$p1 setClass
/Users/scmijt/Apps/nrl/p2ps-ns2/classes pai.examples.ns2.PAICommands"
$ns_ at 0.0 "$p2 setClass
/Users/scmijt/Apps/nrl/p2ps-ns2/classes pai.examples.ns2.PAICommands"

puts "Starting simulation ..."

$ns_ at 0.0 "$p1 javaCommand init"
$ns_ at 0.0 "$p2 javaCommand init"

$ns_ at 0.0 "$p1 javaCommand setSendTo [$n2 node-addr]"

$ns_ at 0.0 "$p1 javaCommand start"

# The timer is started within C++ code NOT Java but the
# parameters are specified here

$ns_ at 0.0 "$p1 startTimer 1 -1"

```

```

# Stop
$ns_ at 9.0 "$p1 stopTimer"
$ns_ at 9.0 "$p2 stopTimer"

#Clean up objects

$ns_ at 10.0 "$p1 cleanUp"
$ns_ at 10.0 "$p2 cleanUp"

$ns_ at 10.0 "finish $ns_"

proc finish {ns_} {
    $ns_ halt
    delete $ns_
}

$ns_ run

```

This example, will run the timer once per second (well, NS2 second anyway - which is non real-time so in effect one second will be microseconds) and iterate for 10 iterations as specified by the NS2 time-stepping as shown.

7.4 Example 3: Sending Data Using Multicast

A Java example, which also implements the ProtoApp scenario but this timer uses a multicast address to send the data between the nodes. The first node sends the data to the multicast address and the second node listens to this address and gets notified when something happens. This example uses the *pai.examples.ns2.MulticastTimerDemo* Java class to implement the Java functionality.

7.4.1 The TCL Side

The following is the TCL script part of the implementation, which creates a muticast enabled NS2 and creates a multicast address for communication. The multicast address to be used must be specified in NS2 and then passed to the Java objects so they know which address to use i.e. by using the *setGroupAddress* java TCL script command as illustrated below. Two *JavaAgent* NS2 nodes are created and attach a *MulticastTimerDemo* object:

```

# Create multicast enabled simulator instance
set ns_ [new Simulator -multicast on]

```

```

$ns_ multicast

# Create two nodes
set n1 [$ns_ node]
set n2 [$ns_ node]

# Put a link between them
$ns_ duplex-link $n1 $n2 64kb 100ms DropTail
$ns_ queue-limit $n1 $n2 100
$ns_ duplex-link-op $n1 $n2 queuePos 0.5
$ns_ duplex-link-op $n1 $n2 orient right

# Configure multicast routing for topology
set mproto DM
set mrthandle [$ns_ mrtproto $mproto {}]
if {$mrthandle != ""} {
    $mrthandle set_c_rp [list $n1]
}

# 5) Allocate a multicast address to use
set group [Node allocaddr]

puts "Creating Java Broker Agents ..."
# Create two Protean example agents and attach to nodes
set p1 [new Agent/JavaAgent]
$ns_ attach-agent $n1 $p1

set p2 [new Agent/JavaAgent]
$ns_ attach-agent $n2 $p2

puts "CREATED OK          ...."

# Initialize C++ agents

puts "In script: Initializing ..."

$ns_ at 0.0 "$p1 initAgent"
$ns_ at 0.0 "$p2 initAgent"

#set up the class

$ns_ at 0.0 "$p1 setClass
/Users/scmijt/Apps/nrl/p2ps-ns2/classes pai.examples.ns2.MulticastTimerDemo"
$ns_ at 0.0 "$p2 setClass
/Users/scmijt/Apps/nrl/p2ps-ns2/classes pai.examples.ns2.MulticastTimerDemo"

puts "Starting simulation ..."

$ns_ at 0.0 "$p1 javaCommand setGroupAddress $group"

```

```

$ns_ at 0.0 "$p1 javaCommand init"
$ns_ at 0.0 "$p2 javaCommand init"

$ns_ at 0.0 "$p1 javaCommand start"

# The timer is started within C++ code NOT Java but the
# parameters are specified here

$ns_ at 0.0 "$p1 startTimer 1 -1"

# Stop
$ns_ at 9.0 "$p1 stopTimer"
$ns_ at 9.0 "$p2 stopTimer"

#Clean up objects

$ns_ at 10.0 "$p1 cleanUp"
$ns_ at 10.0 "$p2 cleanUp"

$ns_ at 10.0 "finish $ns_"

proc finish {ns_} {
$ns_ halt
delete $ns_
}

$ns_ run

```

We then initialize the two *JavaAgent* NS2 nodes start the first node. This results in the first node sending a data packet to the chosen Multicast address, which results in the second node receiving notification of this transfer. The timer is then kicked off, which repeats this process 10 times

7.4.2 The Java Side

On the java side *MulticastTimerDemo.java* implements various commands, rather similar to the *PAICommands* class, except that it replaces the *set-Sender* function with the Multicast address, enabling all nodes to talk to a central address. This enables nodes to automatically send data to collections of nodes and it is this process that will enable P2PS to discover the address of other nodes using its discovery mechanisms. The code looks like this:

```

package pai.examples.ns2;

import pai.broker.CommandInterface;

```

```

import pai.broker.PAIAccessInterface;
import pai.broker.JavaBroker;
import pai.api.PAIInterface;
import pai.net.DatagramSocket;
import pai.net.DatagramPacket;
import pai.net.InetAddress;
import pai.net.MulticastSocket;
import pai.impl.PAITimer;
import pai.impl.Logging;
import pai.event.PAISocketEvent;
import pai.event.PAISocketListener;

import java.net.SocketException;
import java.io.IOException;

/**
 * @author Ian Taylor.
 * A demo of a NS2 Java Object that
 */
public class MulticastTimerDemo implements CommandInterface, PAIAccessInterface,
                                           PAISocketListener {

    PAIInterface pai;
    MulticastSocket s;
    PAITimer t;
    int count=0;

    public void init() {

        try {
            s = new MulticastSocket(5555);
            pai.addPAISocketListener(s,this);
            pai.joinGroup(s,
                new InetAddress(JavaBroker.getMulticastAddress()));
        } catch (SocketException e) {
            System.out.println("Error opening socket");
        }
        catch (IOException ep) {
            System.out.println("Error opening socket");
        }
    }

    void start() {
        timerTriggered(); // transmit first packet right away
    }

    public void dataReceived(PAISocketEvent sv) {
        try {
            System.out.println("Receiving -----");
            ++count;
        }
    }
}

```

```

        byte b[] = new byte[15];
        DatagramPacket p = new DatagramPacket(b,b.length);
        pai.receive(s, p);
        if (Logging.isEnabled()) {
            System.out.println("PAICommands: Received " +
                "PACKET NUMBER -----> " + count);

            System.out.println("PAICommands: Received "
                + new String(p.getData()) +
                " from " + p.getAddress().getHostAddress());
        }
    } catch (IOException ep) {
        System.out.println("PAICommands: Error opening socket");
    }
}

public void timerTriggered() {
    try {
        byte b[] = (new String("Hello Proteus " +
            String.valueOf(count)).getBytes());
        System.out.println("Address is " +
            JavaBroker.getMulticastAddress());
        DatagramPacket p =new DatagramPacket(b, b.length,
            new InetAddress(JavaBroker.getMulticastAddress()), 5555);
        pai.send(s,p);
    } catch (IOException eh) {
        System.out.println("Error Sending Data");
    }
}

public String command(String command, String args[]) {
    if (command.equals("init")) {
        init();
        return "OK";
    }
    else if (command.equals("start")) {
        start();
        return "OK";
    }
    else if (command.equals("trigger")) {
        timerTriggered();
        return "OK";
    }
    else if (command.equals("cleanUp")) {
        pai.cleanUp();
        return "OK";
    }
    return "ERROR";
}
}

```

```

    public void setPAI(PAIInterface pai) {
        this.pai=pai;
    }
}

```

The first thing to notice here is that we are not using the *PAI* Java interface to create our Multicast socket, but we are using the *MulticastSocket* class. The *MulticastSocket* class we are using here is the PAI re-implementation of the `java.io.MulticastSocket` class for use with our Java PAI interface. The actual implementation of *MulticastSocket* simply calls the PAI interface in order to create the appropriate socket, that is in this case, it creates a normal `DatagramSocket` by using the default constructor and sets Multicast to *true* on this socket so that it can join the multicast group address.

The actual multicast group address being used is set from the TCL script, as described. Java NS2 object gain access to this address by using the:

```
JavaBroker.getMulticastAddress();
```

static method call. This enables any Java object within this JVM to gain access to the default Multicast address that it should use. P2PS uses this same address also when communicating with other P2PS nodes, as we will see in Chapt. ???. Here therefore, we join the Multicast group by issuing the following PAI command:

```
pai.joinGroup(s, new InetAddress(JavaBroker.getMulticastAddress()));
```

The rest of the class simply implements the same functionality as the `PAICommands` class discuss earlier in this chapter.

7.5 Conclusion

In this chapter, the Java PAI interface was discussed. An overview of the architecture was given and a brief description of how the classes implement this functionality. We then outlined three examples, which show how one would send data between NS2 nodes, how one would use the timing interface to send repeated calls and how one would use a Multicast address to send data to any nodes that are listening to this address.

Bibliography

- [1] P2PSX: P2PS Experimental for Simulation, see *p2psimplified.org*.
- [2] Touch J (2001) Overlay Networks, *Computer Networks*, 3 (2-3), 115-116.
- [3] Shirky C (2000), Modern P2P Definition, see <http://www.openp2p.com/pub/a/p2p/2000/11/24/shirky1-whatisp2p.html>
- [4] The Protolib toolkit, see <http://pf.itd.nrl.navy.mil/projects/protolib/>
- [5] SAGA Research Group, see <https://forge.gridforum.org/projects/saga-rg/>
- [6] The GAP Interface, see <http://www.gapinterface.org>
- [7] Grid Applications / Grid Application Programming Interfaces Working Group, organised by GRIDSTART, see <http://www.gridstart.org/twg.shtml>
- [8] Allen G, Davis K, Dolkas K, Doulamis N, Goodale T, Kielmann T, Merzky A, Nabrzyski J, Pukacki J, Radke T, Russell M, Seidel E, Shalf J and Taylor I (2003). Enabling Applications on theGrid: A GridLab Overview JHPCA Special issue on Grid Computing: Infrastructure and Applications, August 2003.
- [9] The Workshop on Grid Applications Programming, July 2004, 19-21, EPPC, Edinburgh.
- [10] Ian Taylor, Matthew Shields, Ian Wang, Omer Rana, Triana Applications within Grid Computing and Peer to Peer Environments, Journal of Grid Computing, Volume 1, Issue 2, 2003, Pages 199 - 217.
- [11] The Triana Project, see <http://www.trianacode.org/>

- [12] The Ns2 Simulator, see <http://www.isi.edu/nsnam/ns/>
- [13] The Java Home Page, see <http://java.sun.com/>
- [14] The Java Tutorial: A practical guide for programmers, <http://java.sun.com/docs/books/tutorial/>
- [15] Jxta, see <http://www.Jxta.org/>
- [16] The Gridlab Project, see <http://www.gridlab.org/>
- [17] The GridOneD Project, see <http://www.gridoned.org/>
- [18] Langley, A. The Trouble with JXTA, see http://www.openp2p.com/pub/a/p2p/2001/05/02/jxta_trouble.html
- [19] The log4j logging system, see <http://logging.apache.org/log4j/docs/>
- [20] P2PS: Peer to Peer Simplified, see <http://www.p2psimplified.org/>
- [21] The Protean Research Group, SRSS project, see <http://cs.itd.nrl.navy.mil/5522/>
- [22] The Jini web site: <http://www.jini.org/>
- [23] Gamma E et al. Design Patterns: Elements of Reusable Object-Oriented Software, 1994, publisher Addison-Wesley, ISBN: 0201633612
- [24] UDDI Technical White Paper, UDDI.org, September 6, 2000, see website <http://www.uddi.org>
- [25] Web Services Invocation Framework (WSIF), see website <http://ws.apache.org/wsif/>

Index

Agentj installation, 16

Applet, 54

GAP, 10

GAP Upperware, 11

GAT, 10, 11

Java, 55

JNI, 12, 55

JVM, 55

JXTA, 12

MANET, 6

NS2, 47, 55

Ns2, 41

P2P Discovery, 10

P2PS, 12, 54

PAI, 54

Protolib, 54

Protolib Installation, 15

SAGA Research Group, 10

SRSS Group, 6

UDDI, 12

WSIF, 12